TURING 图灵程序设计丛书

Effective Perl Programming
Ways to Write Better, More Idiomatic Perl Second Edition

Perl高效编程
（第2版）

Joseph N. Hall
[美] Joshua A. McAdams 著
brian d foy
盛春 王晖 张东亮 蒋永清 译

人民邮电出版社
北京

图书在版编目（CIP）数据

Perl高效编程 : 第2版 / (美) 霍尔 (Hall, J. N.) , (美) 麦克亚当斯 (McAdams, J. A.) , (美) 福瓦 (Foy, B. D.) 著 ; 盛春等译. -- 北京 : 人民邮电出版社, 2011.5
（图灵程序设计丛书）
书名原文: Effective Perl Programming: Ways to Write Better, More Idiomatic Perl, Second Edition
ISBN 978-7-115-25046-9

Ⅰ. ①P… Ⅱ. ①霍… ②麦… ③福… ④盛… Ⅲ. ①Perl语言－程序设计 Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2011)第053111号

内 容 提 要

本书是Perl编程领域的“圣经级”著作。它提供了一百多个详实的应用案例，足以涵盖编程过程中经常遇到的方方面面，由此详细阐释出各种高效且简洁的写法。本书第1版曾畅销十年之久，而在第2版中不仅修正了前版存在的一些问题，更与时俱进地引入了许多Perl领域的新主题，使内容更加完善丰富，也更具实用性。

本书为初级Perl程序员铺就了一条通往高阶之路，而对高级Perl程序员来说，本书也是必备的技术参考。

图灵程序设计丛书

Perl 高效编程（第2版）

◆ 著 [美] Joseph N. Hall Joshua A. McAdams brian d foy
译 盛 春 王 晖 张东亮 蒋永清
责任编辑 朱 巍
执行编辑 王一枝

◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街14号
邮编 100061 电子邮件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
北京艺辉印刷有限公司印刷

◆ 开本：800×1000 1/16
印张：20.75
字数：516千字 2011年5月第1版
印数：1－3 000册 2011年5月北京第1次印刷

著作权合同登记号 图字：01-2010-5083号

ISBN 978-7-115-25046-9

定价：65.00元

读者服务热线：(010)51095186转604 印装质量热线：(010)67129223
反盗版热线：(010)67171154

www.TopSage.com

版 权 声 明

Authorized translation from the English language edition, entitled *Effective Perl Programming: Ways to Write Better, More Idiomatic Perl, Second Edition*, 978-0-321-49694-2 by Joseph N. Hall, Joshua A. McAdams, brian d foy, published by Pearson Education, Inc., publishing as Addison Wesley, Copyright © 2010 by Pearson Education, Inc.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

CHINESE SIMPLIFIED language edition published by PEARSON EDUCATION ASIA LTD. and POSTS & TELECOM PRESS Copyright © 2011.

本书中文简体字版由Pearson Education Asia Ltd.授权人民邮电出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

本书封面贴有Pearson Education（培生教育出版集团）激光防伪标签，无标签者不得销售。

版权所有，侵权必究。

www.TopSage.com

大家网
TopSage.com

推 荐 序

十年前，当我开始学习 Perl 的时候，我认为自己对这门语言已经了解得很多了——没错，对这门语言本身，我确实知道得很多。而我所不知道的，则是那些真正赋予 Perl 力量的惯用方法和其他灵活的语法结构。尽管不用它们也能写出绝大多数程序，但不掌握这些，则意味着自己的知识结构还不够完善，自己的工作效率也远远达不到理想状态。

我是幸运的，因为我得到了本书的第 1 版。不过，那本书从来没有机会停留在我的书架上，它一直都在我的包里，一有空我就会打开来读一段。

Joseph N. Hall 这本书的内容编排简单得让人爱不释手，每一段内容虽短，但都饱含智慧，而且讲得十分明白透彻。不瞒您说，我们免费的 Perl Tips 电子报（http://perltraining.com.au/tips/）正是受了本书的启发才创刊的，这份电子报一直致力于探讨 Perl 及其社区的发展。

对于一门语言来说，十年意味着很大的变化，而社区对语言的认知则有更大的变化。因此，让我非常高兴的不仅是听到这本书的第 2 版即将出版的消息，更重要的是这个新版本出自 Perl 社区最杰出的两位成员之手。

不用说，brian 对 Perl 的全心投入是有目共睹的。他不仅写了很多 Perl 语言方面的书，还负责出版一份杂志（*The Perl Review*），并且维护着 Perl 官方网站中的 FAQ（常见问题解答），另外他在众多 Perl 及编程语言社区一直享有盛誉。

而 Josh 则以他运营的著名播客网站 Perlcast 闻名，他从 2005 年就开始在这个网站中以音频形式播放 Perl 新闻了。Josh 总能找到那些著名的、有趣的人，对他们进行采访，这使他自己快速积累了大量知识，也让我对他羡慕不已。

总之，能向亲爱的读者朋友推荐这本书的第 2 版，我感到荣幸之至。希望它能让你真正掌握这门语言的精髓，就像当年第 1 版对我的启蒙那样。

Paul Fenwick
Perl Training Australia 总裁

www.TopSage.com

前　言

很多 Perl 程序员都是通过本书的第 1 版启蒙的。在 1998 年 Addison-Wesley 出版第 1 版的时候，整个世界似乎都在使用 Perl。当时 .com 大潮正在兴起，所有懂点 HTML 的人都能找到程序员的工作。而这些人一旦开始编程，就需要迅速提升自己的技能。本书和其他两本“圣经级”著作 *Programming Perl*[①]、*Learning Perl*[②]基本上是这些新程序员的必读书。

当时市面上还有不少其他的 Perl 书籍。如今的编程学习者应该很难想象当时美国书店的情况，那时候的书店中有数十米的书架摆放的都是编程书，而大多数都是关于 Java 和 Perl 的。如今的书店则只在一个小角落里摆放编程书，每种语言往往只会有几本书，而且大多数的书在上架后的半年内就会被其他书取代。

尽管如此，本书还是畅销了十年之久。这要归功于 Joseph Hall 对 Perl 编程哲学的深刻理解和他本人的过人智慧。毕竟这本书主要讨论的是 Perl 编程思想，他在第 1 版中给出的建议直到现在都还非常实用。

不过，如今 Perl 的世界和 1998 年相比已经有了很大的变化，值得提倡的理念也更多了。CPAN（Comprehensive Perl Archive Network，Perl 综合典藏网）仅仅经过几年的发展，如今已经成了 Perl 最吸引人的特性。人们已经发现了许多更新更好的编程方式，而且这十年来业界积累了更多使用 Perl 的经验，也催生了很多新的最佳实践和惯用技法。

自从本书第 1 版面世以来，Perl 本身也有了很大的变化。第 1 版存在于从 Perl 4 到 Perl 5 的过渡时期，当时大家仍然在广泛使用 Perl 4 的一些古老特性。在这个新版本中，我们基本上消除了这些差异。现在只有一个 Perl，那就是 Perl 5（本书不讨论 Perl 6，那应该另写一本书）。

现代 Perl 已经能够支持 Unicode（而不仅仅是 ASCII），因此你也应该适应这一点，我们为这个主题专门设置了一章。几年来，在 Michael Schwern 的推动之下，Perl 已经成为被测试最多的语言，几乎每一个模块都非常稳定。Perl 粉丝们怀念的“蛮荒时代”已经成为历史。今天，即使是快速原型的开发也可以同时考虑测试。如果你开发的是企业应用，那么你应该好好看一看我们针对测试给出的建议。如果你是一位正则表达式高手，那么你一定想了解最新的 Perl 正则特性，本书将介绍其中那些最常用的。

Perl 仍然在成长中，新的主题还在不断涌现。有些主题本身就值得用一本书的篇幅来介绍，比如 Moose 这个“后现代”的 Perl 面向对象框架，因而本书也就不勉为其难了。另一些主题，

① Larry Wall、Tom Christiansen 及 Jon Orwant 合著的 *Programming Perl, Third Edition*（O'Reilly Media，2000）。

② Randal L. Schwartz、Tom Phoenix 及 brian d foy 合著的 *Learning Perl, Fifth Edition*（O'Reilly Media，2008）。

www.TopSage.com

比如 POE（Perl Object Environment，Perl 对象环境）、对象关系映射器，还有 GUI 工具包等也都因为同样的原因而没有办法在本书中详细介绍。不过，我们已经计划再写一本 *More Effective Perl*，到时候可能会涵盖更多的内容。

最后，Perl 的各种文档和专著较以前也丰富多了。虽然本书会尽可能多地讲到我们认为你应该知道的内容，但如果市面上已经有了详细讨论某些内容的书，我们自然也就不必置喙了，附录会推荐其他一些 Perl 图书，这样做无疑也给更深入地讨论当前本书中的这些主题留出了余地。

Joseph N. Hall、Joshua A. McAdams 和 brian d foy

第 1 版前言

我曾经写过大量 C 和 C++ 代码。在专注于 Perl 之前，我参与的一个重点项目是实现一门脚本语言，该语言能用来画图、计算概率和生成 FrameMaker 格式的书。这个语言用了大约 5 万行平台无关的 C++ 代码，其中有不少相当有趣的特性。应该说，这个项目还是非常有意思的。

它耗费了我两年的时间。

对我来说，大多数有趣的 C 或者 C++ 项目都需要数月乃至数年的时间才能完成。这是因为那些有趣的功能往往比较复杂，所以需要耗费很多时间开发。不过在换为一门高级语言之后，只要 3 个月的时间，我也可以把原先一大堆平淡无奇的想法变成有趣的项目。

这是我最初对 Perl 感兴趣的原因之一。吸引我的是 Perl 这个脚本语言中强大的字符串处理、正则表达式和流程控制能力。这正是那些 C 和 C++ 程序员（面对时间紧凑的项目时）最需要的特性。于是我学了 Perl，而且实实在在地喜欢上了它。这要归功于一个相关项目，我在该项目中负责实现处理文本文件的功能：获取一个程序的输出，格式化之后再发给另一个程序处理。我用 Perl 只花了不到一天就实现了这个功能，而如果使用其他语言，则恐怕需要几天甚至几周时间。

为何要写这本书

我一直都想成为一个作家。小时候我就迷上了科幻小说。我一直热衷于此，有时候一天能读三本，甚至还试着自己写（但写得不好）。之后的 1985 年，我参加了在密歇根州东兰辛市（East Lansing）举办的 Clarion 科幻小说作家培训班。随后一年左右的时间里，我偶尔会写些短篇小说，不过从来没有发表过。后来，上课和工作占用的时间越来越多，我最终也就打消了写科幻小说的念头。不过，我还是坚持写作，只不过写的都是技术文档、教程、方案和文件。当然，我这些年来还陆续接触了好几个技术作者。

其中一个就是 Randal Schwartz。我在一个工程项目中聘用他给我帮忙达一年之久（这是我第一次做技术主管，也是我第一次管理软件开发类的项目，相信大多数认识 Randal 的人能猜到这点）。后来他选择离职去教 Perl，过了一段时间我也去了。

www.TopSage.com

在这段时间中，我对写作的兴趣更浓厚了。在 C++、Perl、Internet 和 World Wide Web 等这些热门领域打拼了很多年，我觉得应该把其中一些有趣的东西写下来。应用和教授 Perl 的经验也让我的这个想法越来越强烈。我盼望着能写一本书，把自己日积月累的各种 Perl 技巧和反反复复遇到的陷阱汇集起来。

1996 年 5 月，在圣何塞的一次开发者大会上，我和 Keith Wollman 有了一次交谈。当时并没有谈到我想写书，我们只是讨论了哪些好的题材可以写成书。当谈到 Perl 的时候，他问我："你觉得一本名叫 *Effective Perl* 的书会不会受欢迎呢？"这个书名打动了我。要知道，Scott Meyers 的 *Effective C++* 是我最喜欢的一本 C++ 著作，而给该系列写一本 Perl 的书显然是个好主意。

Keith 的话始终在我耳边回响。过了一段时间，我在 Randal 的帮助下写了一个选题报告，而 Addison-Wesley 公司批准了这个选题。

接下来，好戏开场了。我开始没日没夜地写作，常常在电脑跟前一坐就是 12 个小时，除了用 FrameMaker 写作，还在 Perl 5 Porters 邮件列表中不厌其烦地问了不少的问题，查阅了几十本书和手册，编写了很多很多段 Perl 代码，也喝了很多很多罐健怡可乐和百事可乐。在查阅资料时，偶尔还会发现一些曾被自己忽略的最基础的 Perl 知识。就这样过了一段时间，本书的第一稿诞生了。

这本书是我的一个尝试，希望借此与大家分享我在学习 Perl 的过程中收获的经验和乐趣。最后，非常感谢你花时间阅读，希望这本书对你有价值，也能让你感到乐在其中。

Joseph N. Hall
亚利桑那州钱德勒市
1998 年

www.TopSage.com

致　谢

第 2 版致谢

许多人帮我审阅第 2 版，指出了我们忽略的一些问题。对此，我们要感谢 Abigail、Patrick Abi Salloum、Sean Blanton、Kent Cowgill、Bruce Files、Mike Fraggasi、Jarkko Hietaniemi、Slaven Rezic、Andrew Rodland、Michael Stemle 和 Sinan Ünür。书中的某些地方也会直接指出他们的贡献。

另外有些人则为我们做了更多，他们几乎针对书中的每一章都提出了问题。正是因为他们的努力，书中许多错误才会在付梓前得以消灭。他们是 Elliot Shank、Paul Fenwick 以及 Jacinta Richardson。若还有错误，或许就只能归咎于我们没有管好自己的猫，这个调皮的小家伙一定是在我们不曾注意的时候到键盘上蹓过弯儿。

Joseph N. Hall、Joshua A. McAdams 和 brian d foy

第 1 版致谢

这本书写来不易。我自认已经倾尽全力了，但如果不是得到了许多程序员、作者、编辑以及其他专业人员的帮助肯定会更加艰难。我衷心感谢所有为本书面市而贡献了宝贵时间和精力的人。

Chip Salzenberg 和 Andreas “MakeMaker” König 帮我修正了不少程序漏洞，使得本书更加精练。对 Chip 的感激光用言语是不足以表达的。我对程序的关注实在太少了，向 Chip 致敬！

Perl 5 Porters 邮件列表工作组也起到了很大的作用，他们为我解答了不少问题。其中尤其要感谢的是 Jeffrey Friedl、Chaim Frenkel、Tom Phoenix、Jon Orwant（以 *The Perl Journal* 杂志[①]闻名）和 Charlie Stross。

Randal Schwartz 是畅销书作者、教师和 Just Another Perl Hacker 的发起人，他也是我最主要的技术审稿人。所以如果你发现书中有任何问题，可以直接给他写邮件（开玩笑的）。非常感谢 Randal，因为他在这本书上付出了许多时间和精力。

感谢 Larry Wall 创造了 Perl，而他本人也解答了我很多的疑问，并且在一些地方提出了建议。

能和 Addison-Wesley 在这个项目上合作，我觉得非常荣幸。这里遇到的每个人都善良而乐于助人，特别要感谢 Kim Fryer、Ben Ryan、Carol Nelson 和 Keith Wollman。

① The Perl Journal 由 Jon Orwant 创立，该杂志已成为联系 Perl 社区的纽带和讨论 Perl 发展的前沿阵地。——编者注

www.TopSage.com

还有许多朋友在其他方面提供了帮助，Nick Orlans、Chris Ice 和 Alan Piszcz 都耗费了大量时间用于阅读初稿。我的几位现任及前任老板 Charlie Horton、Patrick Reilly 和 Larry Zimmerman 则提供了许多灵感，也给了我很大鼓励。

另外，我在写作过程中尽可能坚持原创，但仍不可避免地受到 Perl 在线手册和 *Programming Perl* 等资料的影响。殊途同归，我已尽力用最有创意的方法来阐述，但很多情形下那些有关 Perl 的经典诠释是难以超越的。

非常感谢 Jeff Gong，他总是帮我“骚扰”电话公司，从而让我的 T-1 线路保持畅通。Jeff 总是懂得如何让客户开心。

非常感谢高尔夫这项运动，击球的简单动作能够让我保持清醒，并帮我排解压力。同样地，还要感谢《猎户座之王》和《文明 II》两款游戏。

最后，我必须感谢 Donna，我的未婚妻和终生伴侣，她也是专业的程序员。没有她的支持、鼓励和爱，这本书就无法写成。

Joseph N. Hall
1998 年

www.TopSage.com

引　言

“学习某种语言的基础是一回事，但是知道如何有效使用这个语言进行程序设计则完全是另一回事”，这是 Scott Meyers 在 *Effective C++* 的简介中所说的，这句话对 Perl 来说也同样适用。

Perl 是一门非常高级的语言，可以称为 VHLL（Very High Level Language）。它集成了许多高级语言的功能，比如正则表达式、网络管理和进程管理，而且它的语法也是非常“人性化”的。对于文本处理来说，Perl 比其他常见的计算机语言要好用得多，可以说它是这个领域最好的语言。Perl 对于 Unix 系统管理来说也是极为高效的脚本工具，同时也是 UNIX CGI 脚本开发的最佳选择。Perl 还支持面向对象编程、模块化设计、跨平台开发、嵌入式开发，具有高度的可扩展性。

这本书适合你吗

本书假设你已经有了一些 Perl 开发经验。如果你正准备开始学习 Perl，那么这本书对你来说可能有些早。我们的目标是让你成为一个好的 Perl 程序员，而不只是一个普通的 Perl 程序员。

这本书算不上参考手册，虽然我们确实希望你常常把它放在案头。书中涉及的许多主题比较繁杂，因而我们难以一一深入介绍。不过我们会尝试告诉你最基本的理念，而它们在大多数情况下会很有用。不过这些都还只是敲门砖，你如果真的对此感兴趣的话，还需要更深入地研究一番才行。所以，你仍然需要不时地翻阅一下 Perl 文档，当然也包括附录中提到的那些书。

Perl 有许多值得学的地方

你若读过 Perl 的入门教材，或者学完了相关的课程，那就可以开始写点程序了。不过 Perl 的创造者 Larry Wall 喜欢把这时候写出来的程序称为“咿呀学语”。很多的 Perl 程序都可以归到这一类，它们简单、直白，而且冗长。这不是什么坏事，你可以用适合自己的任何风格来编写 Perl 程序。

但到了某个阶段，你可能会不满于这样的咿呀学语，而希望学习更简洁、更独特的写法。这本书就是为这样的你准备的。它的目标就是让你能写出流畅且表达能力强的 Perl 程序。为此我们会从以下几方面给你一些建议。

- 知识，或者称为“小技巧”。许多很复杂的 Perl 任务其实都有更简单的实现方法，甚至完全有可能只用短短几条语句来实现。Perl 高效编程的秘诀在于积累足够的经验，学会那些做事情的“正确”方法。首先，在看到你认为好的解决方案后，可以用它来解决自己的

www.TopSage.com

问题。其次，当你对什么方案才能算好有了自己的认识后，你也可以自己创造新的方案，然后激扬文字，指出它到底好在哪里。

- 如何使用 CPAN。CPAN 如今已经成了吸引人们学习 Perl 的主要原因。使用其中超过 5G 的源代码、许多重要的编程框架以及各式常用库程序接口，能让你利用他人的成果快速完成许多任务。CPAN 也由此使得常规 Perl 编程任务得以简化。就像学习其他语言那样，你的技能就体现在充分利用已有的成果上。
- 如何解决问题。你可能已从其他语言学会了许多分析问题和调试错误的技巧。这本书通过展示多个问题及其 Perl 解决方案，教你如何使用 Perl 解决问题，同时也通过展示如何高效开发和改善程序，教你如何解决使用 Perl 时遇到的问题。
- 风格。本书主要通过示例来展现 Perl 语言的惯用风格，并教你写出更简洁优雅的 Perl 程序。如果简洁优雅不是你的目标，你至少能了解到如何避免写出某些笨拙的架构。另外，你也能学着品评自己或他人的程序。
- 如何进一步成长。这本书不可能涵盖你想知道的所有东西。虽然我们力图写一本高阶的 Perl 编程资料，但是要想真的涉及所有的高阶主题实在不切实际，因为哪怕只是罗列个大概也可能需要上千页的篇幅。所以本书的主要目标是让你具备成为高级 Perl 程序员的潜质，也就是说使你具备自我提升的能力，这包括如何找到需要的资源，如何不断通过实践来学习，以及如何评断自己的水平。

我们力图把这本书写成一本鼓励思考的书。大多数示例都有不少精微之处，我们会着重阐释那些比较复杂的，而简单的就不再赘言了，我们希望读者自己去领会。有时候我们会特别介绍某段示例代码的一个方面，而对其他部分则略过不表，但是大体上会使其尽可能简单，而又不影响你的理解。如果一时读不懂你也不必紧张，Perl 是一门独特的语言，和你学过的大多数语言都不一样，所以有时候必须得反复练习才能掌握某些窍门。学习的过程固然辛苦，但其间也蕴含着各种各样的乐趣和意想不到的收获。

Perl 的世界

Perl 是一门卓越的语言。在我们看来，它是最成功的模块化编程语言。

事实上，Perl 模块具有“软件芯片”的美名，因为它们常见于各类软件中。（在这里，软件等价于集成电路，是一种可以用于各类应用，而不必理解其内在工作原理的部件。）原因有很多，其中最重要的是因为存在集中协作的模块库 CPAN，它降低了我们在竞争、不兼容功能实现上所消耗的精力。参阅附录可以了解更多资源。

Perl 内建了最基础但足够充分的模块化和面向对象编程框架。由于其中缺乏严格的对象访问限制，Perl 可以写出非常自然简洁的代码。软件行业似乎有个自然法则，那就是最有用的功能却往往来自使用框架和模板最少的开发环境。而 Perl 正是以某种颠覆的方式实现了对这一“规则和惯例”的支持。

Perl 还提供了卓越的跨平台支持。而它之所以适合作为 UNIX 平台的系统管理工具，也正是

www.TopSage.com

TopSage.com

因为它把 UNIX 各个版本之间的差别尽可能地隐藏起来了。你能编写跨平台的 Shell 脚本吗？能，但实在太复杂了。大多数人应该不会进行这样的尝试。你能编写跨平台的 Perl 脚本吗？能，很简单。Perl 程序也可以很好地在 UNIX 和 Windows、VMS 等诸多其他平台间移植。

作为一名 Perl 程序员，你能得到一些世界上最好的资源支持。因为你可以获得所有可用模块的源代码，同时还有语言本身的源代码。你如果觉得自己找出代码缺陷太慢，还可以在网上获得 24/7 的技术支持。如果你不满意免费的技术支持，那么还可以考虑购买商业技术支持。

最终，你有了一门特立独行的语言。Perl 十分流畅，至少在目前几种脚本语言里，Perl 最具表达能力，甚至能达到随心所欲的境界（Do What I Mean，可简称为 DWIM）。这个思路或许有些唬人，却揭示了计算机技术的真正发展进程，它超越了时钟频率、硬盘空间和内存大小等物理性能指标的提升。

术语

通常来说，Perl 语言用到的术语和其他语言中的大致相同，当然一些术语也会有独特的含义。随着 Perl 的发展，其中一些术语已经渐渐不用了，同时也有一些新的术语生成。

通常我们说到语言本身的时候用的是 Perl（P 大写），而说到解释器和源程序的时候用的是 `perl`。大多数时候我们不会特别讨论解释器，所以通常你看到的是 Perl。

操作符（operator）在 Perl 中是一个不需要圆括号的动词（当然它的参数要放在括号中）。

列表操作符（list operator）则是一个标识符，后跟一个以逗号分隔的元素列表：

```
print "Hello", chr(44), " world!\n";
```

Perl 中的**函数**（function）是一个标识符，后跟一对圆括号，其中可以列出所有参数：

```
print("Hello", chr(44), " world!\n");
```

现在你可能看出来了：在 Perl 中列表操作符和函数很类似。实际上，两种语法有相同的功能。一般来说，我们会尽可能使用操作符这个词来描述 Perl 的内置函数，如 `print` 和 `open`，只会偶尔使用函数这个称呼，这两者之间没有本质区别。

在 Perl 中，子程序就叫**子程序**（subroutine），当然称它为“函数”、“操作符”甚至“过程”都是可以接受的。但是请注意，Perl 中“函数”的用法与数学中定义的不同，一些计算机科学家对这个词的滥用颇有微词。

所有的 Perl **方法**（method）都是遵从某种约定的子程序。这些约定既不是 Perl 要求的，也不被正式承认。不过，把这种调用称为“方法调用”还是比较合适的，因为 Perl 为了支持面向对象编程而实现了这种特殊的语法。方法和普通子程序的区别在于：方法的作者期望你按方法调用的语法来呼叫它。

Perl 的**标识符**（identifier）类似于 C 语言，是以字母或者下划线开始，后面跟着一个以上字母、数字或下划线的词。标识符用来命名 Perl 变量。另外，Perl 变量就是组合了某些特定符号的标识符，比如 `$a` 或 `&func`。

www.TopSage.com

虽然我们并不是刻意要使用 Perl 内部的称呼，不过确实可以把 Perl 里面那些具有特别句法意义的标识符称为**关键词**（keyword），比如 `if` 和 `while`。相比之下，其他具有“函数”或“操作符”语法的标识符，比如 `print` 和 `oct`，可以称为**内置函数**（built-in）。

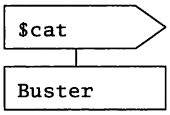
左值（lvalue）是指可以出现在赋值符号左边的值。这是这个词的一般意思，不过对 Perl 来说，某些特别的结构也可以作为左值，比如 `substr` 操作符。

使某个变量**本地化**（localizing）意味着使它的作用域局限在某个代码块或文件的“范围”内。特殊变量必须用 `local` 操作符来实现本地化。而普通变量则可以用 `my` 或者 `local` 来本地化（参考第 4 章的条款 43）。事实上，这是 Perl 的一个设计失误，Larry Wall 当年其实想用另一个名字来代替 `local` 的，但目前只能这样沿用下去了。有时候为了有所区别，我们会明确地说“用 `my` 来本地化”。

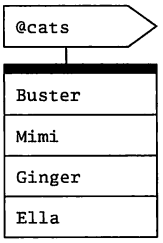
图解

在本书中，我们会使用 Joseph 的 PEGS（Perl Graphical Structure，Perl 图形化结构）来阐明数据结构。这种图示方法非常简单易懂，这里大致介绍一下。

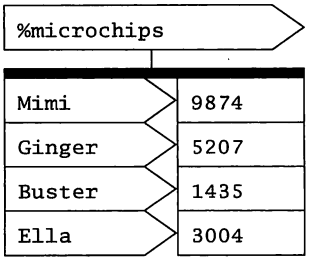
变量是带有名字的值。变量名出现在值的上方，用一侧尖端的方框表示。标量值使用简单的方框来表示：



数组和列表采用相似的图形。其中数值用最上方有粗条的栈表示：

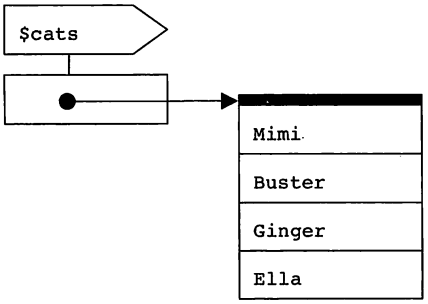


散列则在栈名旁列出相应的值：



www.TopSage.com

引用的画法和 LISP 语言曾使用的图形化方法相似，使用点号加上箭头的方式表示：



以上就是数据结构的所有基本图示方法。

Perl 风格

从这本书中你应该还能领会到一些好的 Perl 编程风格。

当然了，所谓风格也是个人的喜好和考虑。我们不认为自己掌握的是最佳风格，不过还是希望读者会看到一种鲜活、实用而高效的编程风格。不过，有时候为了增加数据的可读性，我们也会暂时牺牲在风格方面的追求。基本上，本书的风格是遵从 **perlstyle** 文档的。

此外，考虑到书中篇幅对代码特有的限制，我们会特别注意行的长度，避免列出那种长篇大段（需要好几页）的代码，并尽可能避免采用那些太过冗长、枯燥乏味的代码作为例子。我们要求每个例子都能突出一两个重点，以避免使读者在代码中迷失。因此，你可能会发现它们不一定遵从最佳实践的要求。

在某些例子中，我们可能会为了突出重点而略过一些细节。在某些代码中，我们使用“...”来代表被省略的代码。（不过，到了本书面世的时候，你可能会发现这个“...”也成了一个合法的语句。因为 Perl 5.12 引入了 yadda yadda 操作符，这使得该语句能成功通过编译，仅在执行时会产生运行时错误。但这是编写桩代码的一种好方法。）

有些例子需要特殊版本的 Perl 才能运行。这里我们如果不特别声明的话，就是需要 Perl 5.8，这个版本稍微有点老，但却非常成熟。如果用到了 Perl 5.10 的特性，那么我们会在例子的第一行注明（例如第 1 章的条款 2）：

```
use 5.010;
```

另外，我们不会介绍 Perl 的开发版本，它们的小版本号一般为奇数，比如 Perl 5.009 或者 Perl 5.011。我们介绍某个特性的时候，会列出引入它的首个稳定版本。

并非每段代码都能通过 `warnings` 或者 `strict` 的约束（见条款 3）。我们建议所有的 Perl 程序员都遵从这两条最基本的约束。不过如果举例时总是提前声明这些约束，也可能会让读者抓不住我们所要论述的重点，因而我们舍弃了这种做法。如果合适，我们会尽可能遵守这些约束，但有时候从大段程序中抽取出来的代码并不一定和原来的程序一样严谨。

我们一般也会尽量减少标点符号的使用（见条款 18）。当然我们不是要鼓励大家尝试“Perl 高

www.TopSage.com

尔夫”[①]，这个游戏关心的是程序最短能写成什么样。我们只是试图去除那些不必要的字符，而更多使用空白，这样能突出真正重要的部分，而不是那些死板的模式。

最后，我们尽可能选取有实用价值的代码。虽然并非所有例子都特别有用，但我们已尽可能采用某些实际应用程序中的代码。

组织结构

前两章主要是为了之后的章节做铺垫，使得读者能够逐渐适应较复杂的编程，而后的内容就会比较有挑战性。阅读过程中，请随时查看本书的目录和 Perl 文档（可以访问 http://perldoc.perl.org/）。

在第 2 版中，我们重新组织了书的结构。我们把第 1 版中某些条款分解成了多个新的条款，同时也对某些进行了合并或删除，因为其他的书中可能已经涉及了这些主题。附录则列出了一些你可能会用到的资源。

这本书的内容并非仅限于这些纸张之上，我们同时还架设了一个网站（http://effectiveperl-programming.com/）。那里你会发现更多与本书有关的信息，有些是我们遗漏的，有些是还未来得及写入本书的，还有些是其他 Perl 相关的有用话题。

① Perl 高尔夫（Perl Golf）是一种不定时举行的 Perl 程序设计游戏，以程序编码最短者获胜，就像高尔夫球中最少杆者获胜一样，详细可参考：http://perlgolf.sourceforge.net/。——编者注

www.TopSage.com

目　录

第 1 章　Perl 基础 ····· 1

条款 1　查阅 Perl 及其模块的文档 ····· 1
条款 2　在需要时开启 Perl 新特性 ····· 3
条款 3　打开约束指令，让编码更规范 ····· 4
条款 4　了解魔符的含义 ····· 7
条款 5　弄清变量名字空间 ····· 8
条款 6　了解字串和数值比较间的差异 ····· 9
条款 7　弄清何时变量为假，并依此作出正确判别 ····· 11
条款 8　理解字符串和数字之间的转换 ····· 14
条款 9　弄清列表和数组间的差别 ····· 17
条款 10　如需空数组，请勿用 undef 赋值 ····· 19
条款 11　仅需单个元素时请勿用切片 ····· 21
条款 12　理解上下文及其对操作的影响 ····· 25
条款 13　用数组或散列集结一组数据 ····· 27
条款 14　用 bignum 处理大数 ····· 29

第 2 章　Perl 的地道用法 ····· 31

条款 15　为优雅、简洁而使用$_ ····· 32
条款 16　了解其他默认参数 ····· 35
条款 17　常见简写和双关语 ····· 37
条款 18　避免过分依赖标点符号 ····· 41
条款 19　调整列表格式以便于维护 ····· 43
条款 20　善用 foreach、map 和 grep ····· 44
条款 21　了解各种字符串引用方法 ····· 47
条款 22　掌握多种排序方式 ····· 50
条款 23　通过智能匹配简化工作 ····· 54
条款 24　用 given-when 构造 switch 语句 ····· 55
条款 25　用 do {}创建内联子程序 ····· 58
条款 26　用 List::Util 和 List::MoreUtils 简化列表处理 ····· 60
条款 27　用 autodie 简化错误处理 ····· 63

www.TopSage.com

大家网 TopSage.com

第 3 章 正则表达式 …… 66
条款 28 了解正则表达式操作符的优先级 …… 66
条款 29 使用正则表达式的捕获功能 …… 69
条款 30 使用更精确的空白字符组 …… 74
条款 31 使用命名捕获，给匹配加标签 …… 77
条款 32 仅需分组时，用非捕获括号 …… 78
条款 33 小心处理匹配变量 …… 79
条款 34 能懒则懒，不要贪婪 …… 81
条款 35 用零宽断言匹配字串中的特定位置 …… 82
条款 36 简单字串处理应避免使用正则表达式 …… 85
条款 37 提高正则表达式的可读性 …… 87
条款 38 避免不必要的回溯 …… 90
条款 39 仅编译正则表达式一次 …… 93
条款 40 预编译正则表达式 …… 94
条款 41 正则表达式的性能评测 …… 95
条款 42 不要滥造正则表达式 …… 97

第 4 章 子程序 …… 99
条款 43 理解 my 和 local 之间的差异 …… 99
条款 44 若非必要请勿直接使用@_ …… 106
条款 45 用 wantarray 按需返回列表 …… 108
条款 46 传递引用而非副本 …… 110
条款 47 用散列传递命名参数 …… 113
条款 48 通过参数原型声明以特殊方式解析参数 …… 116
条款 49 创建闭包锁住数据 …… 118
条款 50 用子程序创建新子程序 …… 121

第 5 章 文件与文件句柄 …… 124
条款 51 不要忽略文件测试操作符 …… 124
条款 52 始终以三项参数的形式调用 open …… 126
条款 53 采用不同方式读取数据流 …… 127
条款 54 处理字符串的文件句柄 …… 129
条款 55 灵活的输出方式 …… 132
条款 56 用 File::Spec 或 Path::Class 处理文件路径 …… 134
条款 57 将数据留于磁盘以节约内存 …… 136

第 6 章 引用 …… 139
条款 58 理解引用和引用的语法 …… 139
条款 59 将引用类型和原型进行比较 …… 145

www.TopSage.com

条款 60　通过引用创建包含数组的数组 …… 147
条款 61　别将匿名数组和列表直接量搞混淆 …… 149
条款 62　通过匿名散列创建 C 风格的 struct 结构 …… 150
条款 63　小心循环数据结构 …… 152
条款 64　用 map 和 grep 操作复杂数据结构 …… 154

第 7 章　CPAN …… 158

条款 65　以非管理员权限安装 CPAN 模块 …… 159
条款 66　拥有自己的 CPAN …… 161
条款 67　减少公共代码带来的风险 …… 164
条款 68　安装模块前先行调研 …… 166
条款 69　确保 Perl 能找到我们的模块 …… 168
条款 70　为 CPAN 作贡献 …… 171
条款 71　了解常用模块 …… 173

第 8 章　Unicode …… 176

条款 72　在源代码中使用 Unicode 字符 …… 177
条款 73　告诉 Perl 该用何种编码方式 …… 179
条款 74　通过代码值或名字输入 Unicode 字符 …… 180
条款 75　位组字串转换到字符字串 …… 182
条款 76　Unicode 字符和属性的模式匹配 …… 185
条款 77　同字素打交道，而不是字符 …… 188
条款 78　谨慎处理数据库中的 Unicode …… 190

第 9 章　软件分发 …… 192

条款 79　用 Module::Build 构建发行版 …… 192
条款 80　不必手工新建软件发行版 …… 194
条款 81　给模块取个好名字 …… 198
条款 82　在代码中嵌入 Pod 文档 …… 201
条款 83　限制我们的发行版用于特定平台 …… 204
条款 84　检查 Pod 文档 …… 206
条款 85　嵌入其他语言代码 …… 209
条款 86　用 XS 链接低级语言，提高运行速度 …… 211

第 10 章　测试 …… 215

条款 87　用 prove 灵活运行测试 …… 215
条款 88　有目的性地运行测试 …… 218
条款 89　用依赖注入避免特殊测试逻辑 …… 220
条款 90　避免给方法引入不必要的东西 …… 222

www.TopSage.com

条款 91 把程序写成模块便于测试 224
条款 92 用虚拟的对象或接口测试 227
条款 93 用 SQLite 创建测试用数据库 231
条款 94 用 Test::Class 编写结构化测试 232
条款 95 项目一开始就准备好测试 235
条款 96 度量测试覆盖率 240
条款 97 把 CPAN Testers 当作 QA 团队 243
条款 98 设置持续编译系统 244

第 11 章 警告信息 250

条款 99 启用警告功能定位可疑代码 250
条款 100 利用词法作用域选择性启用或关闭警告 253
条款 101 用 die 抛出异常 255
条款 102 用 Carp 来获得栈跟踪信息 256
条款 103 正确处理异常 259
条款 104 通过污染检查跟踪危险数据 261
条款 105 对老旧程序启用污染警告 263

第 12 章 数据库 265

条款 106 预备 SQL 语句以复用并节省时间 265
条款 107 利用 SQL 占位符将参数值自动引起 268
条款 108 通过绑定返回列快速访问数据 270
条款 109 复用数据库连接 272

第 13 章 杂项 275

条款 110 编译并安装自己的 perl 解释器 275
条款 111 用 Perl::Tidy 美化代码 277
条款 112 使用 Perl Critic 280
条款 113 用 Log::Log4perl 记录程序运行状态 284
条款 114 明白循环内的数组何时会被修改 289
条款 115 不要用正则表达式提取逗号分隔的字串 290
条款 116 用 unpack 处理固定列宽的数据 291
条款 117 用 pack 和 unpack 对数据作变形处理 293
条款 118 借用 typeglob 访问符号表 298
条款 119 用 BEGIN 初始化，以 END 善后 300
条款 120 用单行 Perl 命令作为迷你程序 302

附录 Perl 资源 307

www.TopSage.com

第1章 Perl基础



如果你有其他语言的编程经验，现在刚刚接触Perl，此时可能还在琢磨和学习Perl的那些神秘特性，那么，从这里开始吧。本章我们会讨论一些基础知识，尤其是那些让Perl编程新手比较头疼的，容易和之前习惯的其他语言相混淆的特性。

举个例子，我们已经知道Perl的变量一般是由$和@这样的符号开头，后面接标识符组成。但你是否知道像$a和@a这样名称相同但类型不同的变量，是否完全独立，互不相干呢？事实上，它们确实完全不相干（见条款5）。

我们知道@a是个数组，但你是否了解$a[$i]和@a[$i]之间的差别？没错，后者是一个数组切片（见条款9）。

我们知道数字0代表假，同样的，空字符串''也代表假，那么包含一个空格的字符串' '是否也代表假呢？很遗憾，它代表真（见条款7）。

由于Perl本身借鉴了许多其他语言，并且创建者Larry Wall本人作为语言学家给Perl添加了许多不寻常的点子，所以Perl会拥有一些有趣而奇怪的特性。无论如何，一旦养成与Perl同步的程序思维，你就能对这些特性产生更直观而自然的理解。

如果你已经是一个经验丰富的Perl程序员，本章内容可以带你回顾一下基础知识。当然，你也可能会在其中发现一些以前不曾了解的有趣细节，或是学到一些以不同角度向同事阐释Perl理论的方式。

条款1 查阅Perl及其模块的文档

Perl本身自带了很丰富的文档，如果把它们都打印出来，恐怕要耗费大量纸墨。我们试图清点过，但数到2000页左右的时候就数不清了（不用担心，我们用虚拟打印机计算页面数量，这样不会对树木造成威胁，挺环保的）。

Perl文档中包含大量有价值的信息，这些文档绝对值得仔细品读，哪怕只花一点功夫，学会如何从这些文档中查找所需的资料也是值得的。要从这些文档中迅速找到需要的内容，除了要具备相关的知识外，得力的工具也必不可少。

1. perldoc文档阅读器

perldoc这个命令能搜索Perl安装目录树中内嵌文档的Perl模块文件（扩展名为.pm）、.pod

www.TopSage.com

结尾的 POD 文档（见条款 82），以及已安装的各种 Perl 工具的文档。该命令会将它找到的文档格式化后显示出来。让我们先从阅读 perldoc 自身的文档开始吧：

```
% perldoc perldoc
PERLDOC(1) User Contributed Perl Documentation

NAME
      C<perldoc> - Look up Perl documentation in pod format.

SYNOPSIS
      C<perldoc> [-h] [-v] [-t] [-u] [-m] [-l]
……省略了剩余内容……
```

一般来说，将文档的名称作为参数传递给 perldoc 命令，即可查阅该文档。比如下面，给定文档名称 **perltoc**，就会显示所有内置文档的目录：

```
% perldoc perltoc
```

你也许会对 **perlsyn** 文档的内容感兴趣，该文档主要描述 Perl 的基本语法：

```
% perldoc perlsyn
```

如果想阅读关于 Perl 内置函数的用法，可以查看 **perlfunc** 文档：

```
% perldoc perlfunc
```

至少得把 **perlfunc** 文档通读一遍，才能大致了解 Perl 都能提供些什么功能。当然，不需要把它们都记下来，只要在大脑中有个印象就够了。比如说，只要知道 Perl 有个内置函数可以处理 /etc/passwd 文件，但具体是哪个，可以到 **perlfunc** 文档里找。要是觉得滚动这么长的页面太过麻烦，只要记得内置函数的名字，就可以用-f 开关指定，直接阅读该函数相关的内容：

```
% perldoc -f split
```

也可以用 perldoc 来阅读模块内嵌的文档，只需提供模块名作为参数即可：

```
% perldoc Pod::Simple
```

如果想了解这个模块安装在什么地方，可以用-l（小写的 L）开关打印该模块的路径：

```
% perldoc -l Pod::Simple
```

如果要查看模块源代码，可以用-m 开关：

```
% perldoc -m Pod::Simple
```

Perl 文档还包括了 FAQ，可以在其中找到许多常见问题的解答。虽然有个在线版本可以阅读，但命令行的 perldoc 有个好处，就是可以用 -q 开关搜索相关主题，非常方便。比如有关随机数的处理，可以试试这个：

```
% perldoc -q random
```

2. 在线文档

在写本书时，http://perldoc.perl.org/是最好的 Perl 在线文档站点。它包含了 Perl 最近几个版本

www.TopSage.com

TopSage.com

的核心文档（包括 HTML 和 PDF 格式），不只如此，它还会记录你曾经看过哪些文档。

但该站点并不提供所有模块的文档。想要看到所有模块的文档，可以到 CPAN Search（http://search.cpan.org/）和 Kobes's Search（http://kobesearch.cpan.org/）这两个站点去看看。这两个站点都提供 CPAN 的网页查询界面。许多人甚至觉得直接访问以上站点，比阅读本机附带的文档还要方便。

CPAN Search 很有用，尤其是它在每个模块页面中都提供了其他工具链接。其中一个工具有类似 `grep` 的功能，用它可以方便地在某个模块不同的发行版本中搜索特定内容。比如想要跟踪某个出错信息来自哪个模块中的哪个文件，就可以用它来试一试。

AnnoCPAN（http://annocpan.org/）是另一个提供模块文档的站点。但它的出发点是让任何人都有机会对模块文档本身添加评注，或是给作者或其他用户留言。这个功能在文档信息不全或者不正确、不完整的情况下显得特别有用。

3. 本地文档

在本地也能够实现部分和 CPAN Search 站点类似的功能。借助 `CPAN::Mini::Webserver` 模块，我们可以在本地搭建一台 Web 服务器，然后通过浏览器查阅自己的 MiniCPAN 库（见条款 66）。如果运行的是 Apache Web 服务器，你还可以用 `Apache::Perldoc` 模块，在 Web 界面中调用本地的 `perldoc` 命令。

此外，`Pod::POM::Web` 模块也能让你以 Web 方式查看本地文档。以 mod_perl 或者 CGI 脚本的形式在 Apache 上运行都没问题，或者干脆直接使用其内置的 Web 服务器：

```
% perl -MPod::POM::Web -e "Pod::POM::Web->server"
```

4. 要点

- 使用 `perldoc` 命令阅读 Perl 文档。
- 本地未安装模块的文档可以通过网页在线阅读。
- 架设本地文档服务器，以 Web 方式浏览本地文档。

条款 2　在需要时开启 Perl 新特性

从 Perl 5.10 开始，新特性必须开启才能使用。这样，新版的 Perl 在让人们享用美妙新特性的同时，还能保持良好的向后兼容。

比如 Perl 5.10 中新加入的内置函数 `say`，它的功能和 `print` 函数几乎一样，唯一不同的是它会自动在每行输出末尾追加一个换行符。使用这个新内置函数不但可以少敲几个字符，而且也不用为了换行符而特意使用双引号圈引整个字串了：

```
say 'Hello!';  # 等效为 print "Hello\n";
```

要是有人创建了和 Perl 5.10 新内置函数 `say` 同名的函数，并在自己的程序中使用该函数，那很可能会因为重名冲突而出错。幸运的是，Perl 默认不启用新特性：

```
% perl5.10.1 say.pl  # 默认不启用新特性
```

```
String found where operator expected at old_script.pl ↵
  line 1, near "say "Hello!""
      (Do you need to predeclare say?)
```

如果想启用新特性，可以使用新的-E开关。它和-e开关类似，一样可以让你在命令行上执行Perl程序，但差别之处在于，-E会打开Perl的所有新特性。

```
% perl5.10.1 -E say.pl   # 打开5.10.1版本中的所有新特性

% perl5.12.0 -E say.pl   # 打开5.12.0版本中的所有新特性
```

你也可以在程序源代码中打开这些特性。只需使用use指令，后接指定的Perl版本号就可以了。这种写法不仅指定了程序能兼容的最低Perl版本（见条款83），还同时打开了该版本中的新特性：

```
use 5.010; # 打开5.10版本以来的新特性
```

从Perl 5.12开始，甚至可以通过直接指定版本号，自动打开约束指令（更高版本的Perl也适用，见条款3）：

```
use 5.012; #自动开启 strict 指令
```

本书中，如果需要用到特定Perl版本的新特性，我们会在示例代码中明确添注上面这行内容。

除此之外，我们还可以用Perl 5.10中新引入的feature这个编译指令打开新特性。给该指令传递一个“特性集合”（feature bundle）参数，告诉Perl载入指定的特性集：

```
use feature ':5.10';
```

如果不希望打开所有新特性，便可用上面提及的feature编译指令，仅打开需要的特性。比如只想打开switch和say这两个新特性，可以这样做：

```
use feature qw(switch say);
```

这里引入的switch好像有些多余，该版本的Perl已经专门提供了一个功能相同、意义明确的关键词given-when（见条款24），不过switch是为了C语言的习惯而延留下来的。

要点

- 从Perl 5.10开始，如果要使用新特性，须输入明确的开启指令。
- 使用-E开关会激活所有新特性。
- 可以用use VERSION语句指定程序运行所需的最低Perl版本，同时还会打开该版本中的所有新特性。

条款3　打开约束指令，让编码更规范

默认情况下，Perl是一门非常自由宽松的语言。我们可以相对随意地组合表达式，Perl知道怎么去理清它们的关系，从而帮助我们减少输入量，这是Perl的一个特点。

但是，Perl这种对编码要求比较宽松的特点，在编写大一些的程序时就不那么讨喜了，它不利于帮助我们编写更规范的代码。但我们可以用strict指令，对Perl编码实行严格要求。只需

www.TopSage.com

在Perl源代码文件中添上这么一句：

```
use strict;
```

如果你使用的是Perl 5.12或更高版本，则可以通过显式指定当前Perl版本号来自动打开约束指令（见条款2）：

```
use 5.012;  # 会自动启用use strict指令
```

通过启用约束指令，编程时常见的错误都会很容易暴露出来。而启用warnings指令（见条款99）的话，还能捕获一些其他不是很紧要的问题。刚开始时你可能会不太习惯，但日积月累能让我们养成好习惯，写出来的代码也会保持规范。一旦成为习惯，以后就不再感到约束，也就不会觉得麻烦了。

如果担心原先的程序启用strict后运行不正常，则可以在实际修改代码前，先在命令行上试着启用strict看看：

```
perl -Mstrict program.pl
```

Perl的约束集包括vars（变量）、subs（子程序）和refs（引用）这三部分。一般情况下，这三种约束会同时使用，稍后我们再分别介绍。

1. 声明变量

Perl程序中最常见的，就是拼写错误。往往是数据存在某个变量，但用的时候却错拼成了其他名字。看下面的例子，文件内容读入@temp后，却在遍历内容时误把数组名写成@tmp，于是你会很困惑为什么不见任何输出呢？有时候无论我们怎么努力检查源代码，就是没法发现问题所在：

```
my @temp = <FH>;
foreach (@tmp) { # 晕，本来是想用@temp的
  print "Found $_\n";
}
```

strict vars编译指令就是专门用来捕获和防止类似错误的，它要求我们事先声明所有用到的变量。变量的声明有三种方式，可以用my，也可以用our：

```
use strict 'vars';

my @temp;
our $temp;
```

或者在声明时使用完整的包名加变量名的形式：

```
use strict 'vars';

$main::name = 'Buster';
```

或者在use vars后面列出变量：

```
use strict 'vars';
use vars qw($bar);
$bar = 5;
```

www.TopSage.com

但无需声明像$_和%ENV这样的特殊变量。此外，strict还会忽略全局变量$a和$b，因为这两个变量是特别设计给sort函数用的（见条款22）。任何其他未声明的变量，都会导致Perl在编译程序时就报错。

2. 小心处理裸字

默认情况下，Perl把不会产生歧义的标识符当成字符串对待（有时这种特性被称作“诗歌模式”），但这很容易导致另一种难以直观发现的错误。你能指出下面代码中的错误吗？

```
for ( $i = 0 ; $i < 10 ; $i++ ) {
  print $a[i];    # 糟糕，其实$a[$i]才正确
}
```

下标i其实应该是$i才对，但在上面的代码中，它被直接写成字符串i，而此时是标量上下文，字符串i会被解释为数字0。因此，这段程序会打印十次$a[0]的值。而启用strict 'subs'后，会关闭诗歌模式，并在发生类似标识符误用的情况下报错。

```
use strict 'subs';

for ( $i = 0 ; $i < 10 ; $i++ ) {
  print $a[i];     # 看到这种写法会报错
}
```

启用strict 'subs'编译指令的情况下，我们仍可以用裸字作为散列的键名。另外，大箭头的左边也是可以用裸字的：

```
use strict 'subs';

$a{name} = 'ok';     # 散列的键可以是裸字
$a{-name} = 'ok';    # 一样可以
my %h = (
   last => 'Smith',  # 裸字在=>符号的左边也可以用
   first => 'Jon'
);
```

3. 避免软引用

strict refs编译指令可以禁止软引用（见条款58）的使用。软引用并不是导致代码产生bug的常见原因，但它也不是经常用的东西。通常在解引用的时候，软引用会跑出来捣乱。比如在下面的例子中，如果没有使用约束，Perl会把变量$not_array_ref内的字符串值当成下面数组变量的名字：

```
no strict 'refs';

$not_array_ref = 'buster';
@{$not_array_ref} = qw( 1 2 3 ); # 误把字串标量当数组引用，但实际会构造一个名为@buster的数组
```

4. 要点

- 默认情况下，Perl是一门很灵活且没有太多约束的语言。
- 不要让Perl来猜你的意图，尽量明确表达你的意思。
- 打开strict开关，让Perl帮你捕获常见的编码错误。

www.TopSage.com

条款4 了解魔符的含义

魔符（sigil）指的是Perl变量名前，或是在解引用时前面用的那个符号。大家总觉得Perl语法妖魅诡惑，大体也都是因为这些符号。但其实有些程序员之所以会觉得迷惑，要么是在不明了的情况下妄自揣测，要么是在邮件列表之类的地方看到了别人对这些符号的胡乱猜测，要么就是在学习Perl的方法上有所偏差。而实际上，正是这些让大家心生畏惧的魔符，反而是实际应用中非常得力的好帮手。

魔符唯一让人迷惑的地方其实是它本身还被用作变量指示符。二者所用的记号虽然相同，但意义截然不同。一旦把魔符和变量的用法和意义分别弄清楚，就不会产生混淆。

$符号代表你操作的是单个值，是一个元素，它可以是一个标量变量，也可以是数组或散列里面的某个元素：

```
$scalar

$array[3]

$hash{'key'}
```

@符号代表你正在操作的是多个值，是一个集合，所以它会和数组及散列用在一起。数组和散列也是Perl中仅有的具备集合性质的数据类型：

```
@array

@array[0,2,6] # 一个数组切片

@hash{ qw(key1 key2) } # 一个散列切片
```

%号有点特别，它代表你正在把某个东西当成散列来操作。而在Perl中，也只有散列这个类型的数据能被这样操作：

```
%hash
```

Perl中还有针对子程序的魔符（&）和针对符号表的魔符（*），但它们分别只针对子程序或符号表这两个数据类型，所以一般不会搞混。

现在我们应该已经知道，不该用魔符作为判断变量类型的因素。可这样的话，又该如何区分呢？有三个考量因素：魔符，标识符，以及数组或散列的索引语法（分别是数字下标和关键字）。我们以获取数组中的单个元素来举例说明：

```
魔符   标识符   索引

 $     name   [3]

$name[3]
```

$是魔符，name是标识符，索引是[3]。因此你可以知道这个变量的名字是name，且从后面的索引语法（使用的是数字下标）来看，很明显就是一个数组。同样，举一个散列的例子：

```
魔符   标识符   索引
```

www.TopSage.com

大家网 TopSage.com

```
$    name  {'Buster'}

$name{'Buster'}
```

可以很容易看出，我们正在操作的是一个散列变量，因为它后面的索引使用了{}，且使用字符串作为索引关键字。$符号只代表你正在操作的变量是散列的一个元素。

如果没有对特定元素的访问下标，基本上仅凭魔符，即可识别变量类型：

```
$scalar

@array

%hash
```

对于匿名的标量、数组和散列，我们也可以用魔符来解引用。至于区分数据类型的规则，同前文所述（见条款59）。

要点

- 魔符只和数据的使用有关，和数据类型无关。
- 魔符$代表的是单个元素。
- 魔符@代表的是一组元素。

条款5　弄清变量名字空间

Perl里面有七种不同的包变量或类似变量的元素：标量变量、数组变量、散列变量、子程序名、格式名、文件句柄及目录句柄。

这些不同的包变量各自都拥有自己的名字空间。改变一种变量的值，并不会影响到同名的另外一种变量。比如说，标量变量$a和数组变量@a是完全独立的两样东西：

```
my $a = 1;              # 赋值，标量变量$a = 1
my @a = ( 1, 2, 3 );    # @a = (1,2,3),但$a仍然为1
```

另外，Perl程序中不同的包都有各自独立的一套名字空间。举例来说，main包里的$a变量和包foo中的$a变量就是两个不同的互相独立的变量：

```
$a = 1;             # 设置标量变量$main::a = 1
package foo;        # 当前包变成foo
$a = 3.1416;        # $foo::a的值为3.1416，而$main::a仍然是1
```

要弄清楚标识符代表什么类型的变量，我们得和Perl一样，考察标识符前后的情形。比如说，要获取数组和散列中单个元素，那么语法上是以$开头的，而不是@或%。请注意，$不是代表标量变量（见条款4），而是说明取一个标量值：

```
my $a = 1;
my @a = ( 1, 2, 3 );
my %a = ( a => 97, b => 98 );
$a[3] = 4;          # $a仍然是1，但@a成了(1,2,3,4)
$a{'c'} = 99;       # $a 和 @a 保持不变
                    # %a现在有三个键值对
```

并不是所有变量类的东西在 Perl 中都是以标点符号开头的。子程序名可以以 & 符号开头，但一般来说这个 & 符号并非必须。甚至在某些条件下，子程序参数的一对圆括号也可以省略，比如下面这个例子。

定义一个名为 hi 的子程序，试试用不同的方式调用它：

```
sub hi {
  my $name = shift;
  return "hi, $name\n";
}
```

我们可以用“老方法”来调用这个子程序，也就是以 & 开头，将参数用圆括号括起来：

```
print &hi("Fred");
```

这种方法如今已不多见。取而代之的，是不使用 & 的简化子程序调用方式：

```
print hi("Fred");
```

圆括号足以让 perl 知道 hi 是一个子程序。要是事先定义或声明过 hi，那么参数两边的圆括号也可以省略：

```
print hi "Fred";
```

文件句柄、格式名及目录句柄都不是以标点符号开头的，所以得靠上下文来区分。文件句柄、目录句柄和格式名彼此之间完全独立，哪怕全都取名为 TEST 也是如此：

```
open TEST, '>', "$$.test"  # 打开文件句柄 TEST
print TEST "test data\n";  # 打印到文件句柄 TEST

opendir TEST, ".";         # 名为 TEST 的目录句柄

# 名为 TEST 的格式
format TEST = @<<<<<<<<<<<<<< @<<<< @<<<<
$name, $lo, $hi
```

有时候，这些东西看起来确实令人头疼。不过还好，现如今有了 IO::File 和 IO::Dir 模块，人们大都改用对象的方式操作文件或目录句柄了。

要点

- 不同的变量类型都拥有独立名字空间。
- 同名但不同类型的变量，彼此之间完全独立，不会互相影响。
- 不同包中的变量可以拥有相同名字，彼此之间不会互相影响。

条款 6　了解字串和数值比较间的差异

Perl 有两套完全不同的比较操作符，一套是用来作字符串比较的，另一套是用来做数值比较的。花些时间弄清楚它们之间的差异还是非常值得的，因为用错比较操作符也许会在代码中留下

www.TopSage.com

比较难发觉的bug。

用来比较字符串的操作符本身也是由字母组成，看起来像是个单词，或者也可以说和FORTRAN里面的操作符比较像。字符串在作比较的时候，是一个字符一个字符按照字符的值（如果你的系统使用了locale，那么会采用locale指定的编码来决定该字符的值）来比较的，并且这个比较的过程会区分大小写和空格，比如：

```
'a' lt 'b'              # 真
'a' eq 'A'              # 假，大小写不同
"joseph" eq "joseph " # 假，空格也是有意义的
```

数值比较操作符的使用，和代数里面的符号差不多，或者说和C语言的差不多：

```
0   < 5       # 真
10 == 10.0    # 真
```

由于字符串比较操作符并不能完全正确地对数字进行比较（除非在你的设定中10在2前面是正确的），所以不能用它们来对数字进行比较。同样的原因，数值比较操作符不能用来比较字符串：

```
'10' gt '2'        # 假，因为'1'排在'2'前面
"10.0" eq "10"     # 假，因为这是两个完全不同的字符串
'abc' == 'def'     # 真，对==来说，它们两个都相当于数字0
```

比较操作符使用不当往往会造成本该为假的比较结果反而成了真，原因就是比较操作符两边的操作数，因为受操作符的影响而被自动转换成了数字：

```
my $hacker = 'joebloe';

if ( $user == $hacker ) { # 错，用==来比较字符串
  deny_access();
}
```

Perl的sort排序操作符默认使用字符串操作符作比较。因此，对数字排序就不能用字串比较操作符（见条款22）。

有种避免把eq和==搞混的方法，那就是避免使用它们。我们可以使用Perl的智能匹配操作符~~（见条款23）来搞定各种比较操作，它会检查符号两边的操作数，然后找出合适的方法来作比较。对于看起来像数字的字符串，这个智能操作符也能在适当的时候把它们当作数字来比较。

下面例子中的比较操作都是数值比较。当智能匹配操作符发现右边是数字，或者发现左边是数字而右边是像数字的字符串的时候，它会按==的方式作比较：

```
use 5.010;

if ( 123      ~~ '456' )    { ... }  # 数字和像数字的字符串：假
if ( '123.0' ~~ 123 )       { ... }  # 像数字的字符串和数字：真
if ( 'Mimi'  ~~ 456 )       { ... }  # 字符串和数字：假
```

否则，智能匹配操作符会按eq的方式来作字符串比较：

```
if ( 'Mimi' ~~ 'Mimi' )    { ... }  # 字符串和字符串：真
```

另外，可能有点诡异的是，如果符号两边都是像数字的字符串，那么智能匹配操作符仍会将

www.TopSage.com

它们当作字符串进行比较（和使用 eq 操作符的效果一样）：

```
if ( '123.0' ~~ '123' )    { ... }  # 像数字的字符串和像数字的字符串：假
```

在对变量使用智能匹配的时候要小心，因为前期对变量的一些操作有可能影响到智能匹配符的判断。如果你之前对 $var 的操作触发了类型转换（见条款 8），那么就有可能因为智能匹配符选用了错误的比较类型而造成非预期的结果。在下面的例子中，一开始使用的是字符串，但因为接下来对它使用了数值操作符，便暗中转换成了数字，最终导致比较操作按 == 的方式进行：

```
use 5.010;

if ( ( '123' + 0 ) ~~ '123.0' ) { # 数字和像数字的字符串：真
  say "Matched 1!";
}

my $var = '123';
if ( ( $var + 0 ) ~~ '123.0' ) { # 数字和像数字的字符串：真
  say "Matched 2!";
}
my $var2 = '123';
$var2 + 0;
if ( $var2 ~~ '123.0' ) { # 数字和像数字的字符串：真
  say "Matched 3!";
}
```

此外，如果一开始是数字，但某些操作导致它被转换为字符串后，Perl 只会把它当作字符串来比较，即按 eq 的方式：

```
use 5.010;

my $var3 = 123;
$var3 = "$var3";
if ( $var3 ~~ '123.0' ) { # 字符串和像字符串的数字，等效于 eq 比较，所以为假
  say "Matched 4!";
}
```

要点

- 字串和数字的比较和排序方式，是有差异的。
- 比较字串要用字符串比较操作符[①]。
- 比较数字要用数值比较操作符[②]。

条款 7　弄清何时变量为假，并依此作出正确判别

因为数值和字符串类型的数据在 Perl 中都属于标量，而布尔操作符能针对任何标量进行操作，所以 Perl 的逻辑真假值测试，可以根据数字和字符串的取值判别。

基本的判别如下：0，'0'，undef，以及''（空字符串）都是假值。其他皆为真。

① 字串比较操作符，本身就是字串。

② 数值比较操作符，本身就是数学记号。

www.TopSage.com

更确切地说，就是在**布尔上下文**（Boolean context，常指判断表达式中作条件判断的部分，比如 ?: 操作符，|| 及 && 操作符）中使用某个定量，它首先会被转换成一个字符串（见条款 8），然后 Perl 会对这个字符串进行判断并得出结果。如果结果是空字符串，或者单单由 '0' 构成的字符串，则判别结果为假。反之为真。请注意，在此规则中，undef 会被当作假，因为除了 defined 操作符，它对任何东西来说都等同于 0 或空字符串。

以上规则对一般情况都适用。如果出现问题，那多半是因为误把返回结果的值作为是否成功返回数据来判断真假：

```
while ( my $file = glob('*') ) { # 这段代码实际上是有问题的
  do_something($file);
}
```

以上代码中的 glob 多数时间都能良好工作。每次循环过程，glob('*')都会得到当前目录中下一个不同的文件名，然后赋值给变量 $file。当 glob 读完了当前目录下所有文件名之后，它会返回等同于空字符串的 undef 表示结束（这就等同于空字符串，为假值），于是 while 循环终止。

但这里有个问题。如果当前目录下恰好有个名为 0 的文件，因为它也是假值，所以会导致 while 循环提前退出。为了避免这种错误，我们应该用 defined 操作符来测试某个值是否已定义：

```
while ( defined( my $file = glob('*') ) ) { # 这才正确
    do_something($file);
}
```

这种情况对于行操作符 <> 也适用，尽管对于这个操作符，Perl 会自动帮我们做类似上例的判断，等效于在 while 循环中对 STDIN 的每一行输入进行是否已定义的测试：

```
while (<STDIN>) {
  do_something($_);
}
```

以上代码就是这个特例，Perl 会自动对 $_ 的值作定义与否的测试，它等效于：

```
while ( defined( $_ = <STDIN> ) ) { # 暗地里自动帮你做了
  do_something($_);
}
```

我们可以使用 B::Deparse 模块来验证这个说法，它可以对 perl 代码进行反编译，这样我们就可以看出代码在底层实际的运行状态：

```
% perl -MO=Deparse \
     -e 'while( <STDIN> ) { do_something($_) }'
while (defined($_ = <STDIN>)) {
     do_something($_);
}
-e syntax OK
```

如果想将当前读到的某行赋值给其他变量，那我们得自己做 defined 检查：

```
while ( defined( my $line = <STDIN> ) ) {
  do_something($line);
}
```

www.TopSage.com

TopSage.com

1. 数组的尾部

当出现一个数组的某个元素未被定义的情况时，并不意味着你对这个数组的访问已经越界了。通常情况下，Perl 对于越界访问很宽容，而这对很多其他语言来说是不可思议的：

```
my @cats = qw( Buster Roscoe Mimi );
my $favorite = $cats[8]; # cats 数组没有这个元素，所以值为 undef
```

数组中的某个元素值为 undef 也没什么大不了的：

```
my @cats = qw( Buster Roscoe Mimi );
$cats[1] = undef; # Roscoe 被换成了未定义的值 undef
```

所以判断是否已遍历了整个数组，不能用数组元素的值来判断：

```
while ( defined( my $cat = shift @cats ) ) { # 错误
  print "I have a cat named $cat\n";
}
```

可以用 foreach 函数来遍历数组的元素，并跳过未定义的值：

```
foreach my $cat (@cats) {
  next unless defined $cat;
  print "I have a cat named $cat\n";
}
```

不要把 undef 当成数组的最后一个元素，应该使用 $#cats 的写法来指代该数组的最后一个元素下标。

```
for ( my $i = 0 ; $i <= $#cats ; $i += 2 ) {
  next unless defined $cat[$i];
  print "I have a cat named $cat[$i]\n";
}
```

2. 散列值

为了测试某个元素是否存在于散列中，我们会用到不同的测试方式，因为未定义的值 undef 同样也是合法的散列值。假设 %hash 未定义，是个空散列。分别用如下的方式测试键为 foo 的元素是否存在：用 defined 函数测试该元素的值；用 exists 函数测试是否存在该键名：

```
my %hash;

if ( $hash{'foo'} )          { ... }  # 假
if ( defined $hash{'foo'} ) { ... }  # 也是假
if ( exists $hash{'foo'} )   { ... }  # 还是假
```

一旦给某个键赋了值，哪怕这个值代表假或者未定义，这个键也就存在了：

```
$hash{'foo'} = undef; # 赋值为 undef，即未定义的值

if ( $hash{'foo'} )          { ... }  # 还是假
if ( defined $hash{'foo'} ) { ... }  # 仍然是假
if ( exists $hash{'foo'} )   { ... }  # 现在为真了

print keys %hash;                    # ('foo')
```

如果某个键的赋值明确，哪怕是个代表假的值，这个键对应的值也是已定义的：

```
$hash{'foo'} = '';

if ( $hash{'foo'} )          { ... }   # 仍然为假
if ( defined $hash{'foo'} ) { ... }   # 现在为真了
if ( exists $hash{'foo'} )   { ... }   # 现在为真了
```

在测试散列元素前，先弄清楚你究竟要测试什么（是否存在，是否已定义，还是值的真假）。用什么样的方式测试，取决于你对值要做什么样的处理。

3. 要点

- 四个代表假的值：undef，''，数字0，字串'0'。
- 除了上面四种情况外，其他值都代表真。
- 务必搞清楚你要测试的值的类型，别只看值的真假。

条款8　理解字符串和数字之间的转换

Perl的标量变量既能存储字符串，也能存储数字。或者说，这两种类型在标量变量中是同时存在的，当字符串和数字间相互转换时便是如此。

必要的时候，Perl会自动将数字转换成字符串来表示，反之亦然。比如若有字符串出现在数值操作符（像+这样的）旁边，Perl在做算数运算前，会先把它转换成数字。如果某个数字是模式匹配中的一部分，Perl则会先将这个数字转换成字符串。

Perl操作中期望出现字符串的地方被称为**字符串上下文**（string context），同样，期望出现数字的地方称为**数值上下文**（numeric context）。我们应该弄清楚这两个术语，但本书中我们不会频繁用到。

Perl用来做数字到字符串转换的函数是和C语言标准库中的sprintf()同名的函数，并同样使用"%.20g"这样的格式化参数。因此对于特定格式，可以用Perl的sprintf函数：

```
my $n = sprintf "%10.4e", 3.1415927; # "3.1416e+00"
```

Perl用来做字符串到数字转换的函数是C语言标准库中的atof()函数。而且转换时，该函数会忽略字符串前导空白字符。Perl做转换时，只关乎字符串前面像数字的部分，其他部分忽略。如果字符串不包含像数字的字符，那这个字符串会被直接当作0处理。请看以下示例：

```
my $n = 0 + "123";          # 结果是123
my $n = 0 + "123abc";       # 结果也是123，尾部的非数字字符会被忽略

my $n = 0 + "\n123";        # 结果也是123，前面的白字符被忽略
my $n = 0 + "a123";         # 结果是0，因为它并非以数字开头
my $n = 0 + "\x{2165}";     # 结果0，因为我们说的数字并不包括罗马数字
```

转换过程也不会识别八进制或十六进制表示的字符串，如果确实需要当作数字处理，可以先用oct操作符转换为十进制：

```
my $n = 0 + "0x123";        # 结果是0，无法理解的字串会被当作0
```

www.TopSage.com

```
my $n = 0 + oct("0x123");  # 结果是 291，oct 函数已将八进制或十六进制转换成十进制

print "mode (octal): ";    # 提示输入文件权限模式
chmod <STDIN>, $file;      # 错误，从标准输入 STDIN 读入的是十进制值，而非八进制值

print "mode (octal): ";    # 提示输入文件权限模式
chmod oct(<STDIN>), $file; # 正确，代表模式的字符串被当作八进制使用
```

当 Perl 自动进行数字到字符串的相互转换时，会保留转换前后的表示方式（数字或字符串），直到变量的值被改变。

1. 同时代表字符串和数字

一般情况下，某个变量包含的是字符串还是数字，都无所谓。但确实还是有需要注意区分的场合。比如位操作符在操作数字时，是完全当它是数字来运算的，而在操作字符串时，则会按字符串的二进制数字运算：

```
my $a = 123;
my $b = 234;
my $c = $a & $b; # 结果为数字 106

$a = "$a";
$b = "$b";
my $d = $a & $b; # 结果为字符串 020
```

可以用 Devel::Peek 模块查看运行时的详细情况。请看下面的例子，字串的部分内容将被转为数字，转换后参与数值运算（但不修改该字串的值）：

```
use Devel::Peek qw(Dump);

my $a = '12fred34'; # 当作数字时，会被转换成 12

print STDERR "Before conversion: ";
Dump($a);

my $b = $a + 0;

print STDERR "After conversion: ";
Dump($a);

print STDERR "\n$a\n";
```

Dump 函数会一一呈现 Perl 内部的追踪细节，虽然我们平时并不需要关心，但这里为了说明实际发生的状况，大致讲解一下。可以看到，转换前 Perl 将实际字符串值保存在 PV 段，而在转换后，得到的数字会保存在 Perl 的 IV 和 NV 段：

```
Before conversion: SV = PV(0x801038) at 0x80e770
  REFCNT = 1
  FLAGS = (PADMY,POK,pPOK)
  PV = 0x204c10 "12fred34"\0
  CUR = 8
  LEN = 12

After conversion: SV = PVNV(0x8023c0) at 0x80e770
```

www.TopSage.com

```
  REFCNT = 1
  FLAGS = (PADMY,IOK,NOK,POK,pIOK,pNOK,pPOK)
  IV = 12
  NV = 12
  PV = 0x204c10 "12fred34"\0
  CUR = 8
  LEN = 12
```

表示错误信息的$!变量，是少数几个具有魔力属性的变量之一。在数字上下文中，它会返回以数字代表错误类型的系统变量errno的值；但在字符串上下文中，它会返回perror()函数（或其他系统中等价的函数）提供的字符串错误信息：

```
open "";                # 非法文件名，理应报错
print "$!\n";           # 字符串上下文，打印错误消息： No such file or directory
print 0 + $!, "\n";     # 数字上下文，打印数字2（或其他表示错误类型的数字）
```

由于given-when语句（见条款24）隐式使用了智能匹配操作符（见条款23），所以它同样会因上下文不同而导致内部转换不同。而智能匹配操作符会根据条件比较的类型作出相应的判断，所以有可能产生意想不到的结果。在下面的例子中，我们试图用智能匹配操作符测试两个字符串，本希望它能匹配完全一样的字符串：

```
use 5.010;

for my $s ( '137', ' 137' ) {
  given ($s) {
    when ('137') { say "$s matches '137'"; continue }
    when (' 137') { say "$s matches ' 137'" }
  }
}
```

程序的输出记录说明，它只会在与给定字符串完全一样的情况下才会匹配：

```
137 matches '137'
 137 matches ' 137'
```

但对程序略作改动，结果就大不一样了。只要该字符串参与过数值运算，就算未曾修改该字符串变量本身，智能匹配操作符也会用数值方式比较：

```
use 5.010;

for my $s ( '137', ' 137' ) {
  my $t = $s + 0;
  given ($s) {
    when ('137') { say "$s matches '137'"; continue }
    when (' 137') { say "$s matches ' 137'" }
  }
}
```

现在的匹配结果和原先预想的不同，多了两条意料之外的记录：

```
137 matches '137'
137 matches ' 137'
 137 matches '137'
 137 matches ' 137'
```

www.TopSage.com

2. 创建自己的双值变量

并非只有靠 Perl 内部转换才能使用双值变量（dualvar）——在不同上下文环境中拥有不同类型值（字符串和数字）的变量。下面的子程序会返回一个普通的标量变量，但它同时拥有数字错误代码和字符串出错信息：

```
use Scalar::Util qw(dualvar);

sub some_sub {

  # ...
  if ($error) {
    return dualvar( -1,
      'You are not allowed to do that' );
  }
}

my $return_value = some_sub();
```

$return_value 变量同时保存有表示错误信息的数字和字符串，除非你另行修改，否则它会保持不变。不过，这是一个比较特殊的功能，就算知道也不应该到处使用，以免程序出现意外结果。留着等到特别的时刻再拿出来，比如说在鸡尾酒会上，或者开发者大会上，雷他们一下。

3. 要点

- Perl 会根据上下文决定将标量当作字符串还是数字。
- Perl 会根据最合理的推测，将字符串转换成数字。
- 不像数字的字符串在转换时会被 Perl 当成 0。

条款 9　弄清列表和数组间的差别

列表和数组是两个不同的概念。看到某个列表或数组时，若能立即分辨清楚，并对相应的规则稔熟于胸，那你的水平恐怕已经高出很多人了（就这点上，甚至比某些 Perl 方面的专业书还要高）。许多 Perl 的强大习语，就是依赖于这种细微差异而衍生出来的。

所谓列表，就是一组有序的标量集合。我们可以自己创建一个列表，或者从内置函数或子程序的返回值中获取一个列表，抑或是从别的列表中抽取一个列表出来。

所谓数组，则是一个存储着列表的变量。数组并非列表本身，而是列表的容器。

因为某些操作的语法格式相同或相近，人们常常分不清楚当下的是列表还是数组。例如，下面这两行代码，都是用相同语法抽取五号元素，但前者是列表，后者是数组：

```
(localtime)[5];  # 该元素为表示年份的数字

$array[5];
```

我们也可以用相同的语法对某个列表或数组进行切片：

```
(localtime)[ 5, 4, 3 ]; # 年，月，日
```

www.TopSage.com

大家网 TopSage.com

```
@array[ 5, 4, 3 ];
```

1. 逗号操作符

对于数组，我们可以将它放在标量上下文中计算，**但对于列表，则没有相应的概念。**标量上下文中的数组，返回的是数组中的元素个数：

```
my @array = qw( Buster Mimi Roscoe );
my $count = @array;                    # 3
```

而列表永远就只是列表，它可没有标量解释。想想你是如何创建一个列表的，用逗号操作符（你没看错，逗号也是个操作符）分隔一系列标量：

```
( 'Buster', 'Mimi', 'Roscoe' )
```

那它现在就是个列表了吗？还不能完全肯定。我们得把这一系列标量用到期望得到列表的表达式中。比如说将它赋值给一个数组，或者用到提供列表上下文的 foreach 循环中：

```
my @cats = ( 'Buster', 'Mimi', 'Roscoe' );

foreach ( qw( Buster Mimi Roscoe ) ) {
  ...;
}
```

（qw() 不会改变任何东西，效果和 @cats 赋值的那个列表相同，只是让语法更简洁。）

如果把同一段代码赋值给标量变量会怎样？得到的会是一个列表吗？$scalar 最终的赋值会是什么呢？

```
my $scalar = ( 'Buster', 'Mimi', 'Roscoe' );
my $scalar = qw( Buster Mimi Roscoe );
```

很多人会猜 $scalar 的值是 3，因为那个列表有三个元素。但赋值符号右边的，并不是作为集合的列表，而是用逗号操作符分隔开的一串标量而已。而逗号操作符在标量上下文中会返回最右边的元素。因此，Perl 会将值 Roscoe 赋给 $scalar。

大家有时候会被这个概念搞迷糊，原因可能在于测试方法不正确。试试这个常见列表？

```
my $scalar = ( 1, 2, 3 );
```

很遗憾，在这个例子中，由于右边标量的个数恰好和列表最右边元素的值相同，此时我们得到的并非列表元素个数，而是逗号操作符返回的最后一个元素。

2. 列表上下文中的赋值操作

Perl 中另一个比较隐晦的操作符就是赋值操作符。很多人自以为对它很了解，但其实并不清楚赋值操作符和其他操作符一样，也会有结果返回。在列表上下文环境中，赋值操作符会返回右边元素的个数。它不会用这个结果来赋值；这个结果只是作为操作符的返回值而已，极少被用到，比如：

```
my $elements = my @array = localtime;
```

由于赋值操作符是右结合的，所以最右边的赋值操作符会先起作用，将 localtime 函数返回的列表赋值给数组 @array。然后赋值操作的返回值，也就是元素个数，会传给变量 $elements。

www.TopSage.com

如果加上圆括号明确标出操作的先后顺序的话，看起来就像这样：

```
my $elements = ( my @array = localtime );
```

这里起作用的原则是，在标量上下文环境中的列表赋值操作，会返回赋值操作符右边的元素个数。相关内容可以到 **perlop** 文档阅读。这个原则非常重要，让我们再重复一遍：标量上下文环境中的列表赋值操作，会返回赋值操作符右边元素的个数。

知道了这个原则，使用的时候便能游刃有余。如果我们想算一下某个操作生成的列表中有几个元素，那就可以像之前 localtime 的例子一样，先把它赋值给一个列表，然后再赋值给一个标量。并且由于这个规则是右结合的，所以实际上中间被赋值的列表可以是个空列表：

```
my $elements = () = localtime;
```

有时候它也被戏称为 **goatse 操作符**，因为写的时候若去掉圆括号两边的空格，中间的部分看起来就像是山羊头[①]（或其他类似的东西）：

```
my $elements =()= localtime;
```

如果要统计全局匹配的个数，或者 split 函数返回的结果有多少元素，就可以利用这个特性。不但方便，而且无需触及原始元素。只需先将结果赋给空列表，然后将赋值操作的返回值保存下来：

```
my $count =()= m/(...)/g;

my $count =()= split /:/, $line;
```

3. 要点

- 列表是一组标量的集合。
- 数组是保存列表的容器，它的本质是变量。
- 列表和数组的多数操作虽然相似，但却有本质的区别。

条款 10　如需空数组，请勿用 undef 赋值

Perl 里面未初始化的标量变量的值为 undef，这个特殊的值代表变量尚未赋值。你可以给变量赋值 undef，或者使用 undef() 函数重置标量变量，使它们恢复到未初始化的状态：

```
my $toast = undef;
undef $history;
```

未初始化的数组变量，其值相当于一个空列表 ()。如果把 undef 赋值给一个数组变量，实际上你得到的会是一个包含单个元素（值为 undef）的数组。所以不要将 undef 赋值给数组变量！在 Perl 中，只要某个列表中含有单个元素，哪怕这个元素是 undef，它也会被 Perl 当成真值，这会在程序中造成难以察觉的问题：

```
@still_going = undef;
if (@still_going) { ... }
```

① 山羊的英文是 goat。——编者注

www.TopSage.com

为了避免发生这种问题，最简单的办法是清空数组变量，只给它赋值空列表 ()：

```
my @going_gone = ();
```

或使用 undef 函数：

```
undef @going_gone;
```

区分 undef 还是 0 或空字符串 '' 的唯一方法，还是使用 defined 操作符。defined 操作符可以针对任意值使用，若变量的值是定义过的，则返回真：

```
if ( defined($a) ) { ... }
if ( defined(@a) ) { ... }
```

你可以将 undef 赋值给数组的元素，但这个操作不会改变数组大小，它只会替换掉某个或某组元素的值，可能会创建一个潜在的稀疏数组：

```
$sparse[3] = undef;
@sparse[ 1, 5, 7 ] = ();
```

给数组的元素赋未定义的值会吃掉一些内存：

```
@sparse[ 0 .. 99 ] = ();
```

请注意，undef 作为元素的值是非常合理合法的。但若试图减小这个数组的大小，而给数组最后一个元素赋值为 undef 是无效的：

```
my @a = 1 .. 10;
$a[9] = undef;
print scalar(@a), "\n";  # 输出"10"
```

在不给整个数组重新赋值的条件下，要减小数组大小，必须得使用专门的数组操作符，比如可以用 pop 函数：

```
my $val = pop @a;
print scalar(@a), "\n"  # 输出"9"
```

或者用 splice 函数：

```
splice @a, -2;
print scalar(@a), "\n";  # 输出"7"
```

或者通过赋值来修改变量$#array_name：

```
$#a = 4;
print scalar(@a), "\n";  # 输出"5"
```

和数组一样，你也不能通过将 undef 赋给某个散列，而将它清空成未定义的状态。实际上，若是将奇数个元素列表赋值给散列表，会得到一个警告信息：

```
Odd number of elements in hash assignment ↵
  at program.pl ...
```

但可以将空列表 () 赋值给某个散列变量，以此创建一个空散列：

```
%gone = ();
```

www.TopSage.com

```
if ( keys %gone ) {
  print "This will never print\n";
}
```

或者，干脆用 undef 函数来重置散列：

```
my %nuked = ( U => '235', Pu => 238 );
undef %nuked;
if ( keys %nuked ) {
  print "This won't print either\n";
}
if ( defined %nuked ) {
  print "Nor will this\n";
}
```

和数组一样，你也不能通过将值 undef 赋给元素以减小散列大小或者移除散列中的元素。如果想移除散列中的元素，得使用 delete 操作符。该操作符可作用于散列的单个元素，或是某个散列切片：

```
my %spacers = (
  husband  => "george",
  wife     => "jane",
  daughter => "judy",
  son      => "elroy",
);

delete $spacers{'husband'};

if ( exists $spacers{'husband'} ) {
  print "Won't print because 'husband' is gone\n";
}

delete @spacers{ 'daughter', 'son' };
```

要点

- undef 是个标量值，且仅对标量变量有效。
- 将 undef 赋值给数组试图使之清空是错误的做法，这只会创建包含单个元素的列表。
- 用 exists 操作符测试某个键是否存在于散列表，而非根据键值的真假作判断。

条款 11　仅需单个元素时请勿用切片

@a[1]是一个数组元素还是数组切片？它就是一个切片。对 Perl 初学者来说，让他们觉得违反常理的一个东西就是数组元素和数组切片之间的差异。就算你已经了解这种差异，往往也会意外地误将 @ 替代 $。

一些介绍 Perl 的书籍或课程，一开始可能都会告诉你标量变量的名字是以 $ 开头的，而数组变量的名字是以 @ 开头的。这种说法虽然没错，但过于简单了（见条款 4）。

为访问数组 @a 的第 $n 个元素，该使用的语法是 $a[$n]，而不是 @a[$n]。看起来有些奇怪，不是吗？尽管如此，这仍是个固定的语法。**标量值（这里说的不是标量变量），是以 $ 开头的，**

www.TopSage.com

不管这个值来自哪里，数组也好，散列也好，都仅仅是一个标量值。

因此，@a[$n]代表的并不是数组@a的第$n个元素，而是称作**切片**（slice）的东西。切片是一种同时访问多个元素的快捷方式。借助切片，就不用重复指定同一个变量名加上不同下标来取得多个元素，如：

```
my @giant = qw( fee fie foe fum );
my @queue = ( $giant[1], $giant[2] );
```

用切片来完成同样的功能，简单易懂：

```
my @giant = qw( fee fie foe fum );
my @queue = @giant[ 1, 2 ];
```

你甚至可以用一个数组来指定需要的下标：

```
my @fifo = ( 1, 2 );
my @queue = @giant[ @fifo ];
```

现在我们知道，@a[1]和@a[1, 2]、@a[2, 10]、@a[5, 3, 1]、@a[3..7]等一样，本质上都是切片：@a[1]是一个只包含单个元素的列表，它并不是标量。

我们要小心单个元素的切片。如果使用方式不对，有可能会造成麻烦。在标量上下文中的切片，返回的是切片中最后一个元素。因此，单个元素切片的作用就像普通标量一样：

```
my $jolly = @giant[3];
```

本意可能是my $jolly = $giant[3]。但这里代表单个元素的切片@giant[3]也行得通，因为@giant[3]在标量上下文中等效于它的最后（也是唯一）一个元素，即$giant[3]。

尽管单个元素的切片和单个数组元素处于赋值操作右边时，表现和效果都差不多，但若放在赋值操作左边时，则会完全不同。因为单个元素的切片实际上是一个列表，对单个元素切片的赋值操作实际上是一个列表赋值操作，因此，赋值操作的右边会被当作是一个列表上下文环境。

如果我们打开了warnings开关（为什么不？见条款99），Perl会对此发出警告：

```
use warnings;

my @giant = qw( fee fie foe fum );

my $jolly = @giant[3];
```

虽然操作的结果和我们预期的一样，将fum赋值给了$jolly变量，但这不能成为日后滥用的借口。由于上下文不一致，Perl会发出一条警告信息，提醒你可能无意中使用了一个切片：

```
Scalar value @giant[3] better written as ↵
  $giant[3] at ...
```

1. 左值切片

在赋值操作中，如果左边是一个切片，那就相当于对这个切片中的变量元素依次赋值。像这种需要用到左边赋值的地方，都可以使用切片。写法上更轻便，而且等效于使用具名变量：

```
( $giant[1], $giant[2] ) = ( "tweedle", "dee" );
@giant[ 1, 2 ] = ( "tweedle", "dee" );
```

www.TopSage.com

TopSage.com

在列表上下文中的操作符有时可能会产生意外甚至不幸的结果。行输入操作符就是个典型的例子，如果赋值给切片，或者只要是在列表上下文赋值，都会导致立即读取整个标准输入 STDIN 中的内容：

```
@info[0] = <STDIN>;
( $info[0] ) = <STDIN>;
```

上面两行代码会从标准输入中读取所有行，然后把第一行存入数组 @info 下标为 0 的元素，而其余行的内容会被抛弃！将 <STDIN> 赋给 @info[0] 会将 <STDIN> 置于一个列表上下文的环境。在这个上下文中，<STDIN> 会从标准输入读取所有行，并将它们以列表形式返回结果。

2. 不要混淆切片和元素

切片和元素之间的区别还有一点，那就是元素方括号内的表达式是在标量上下文中的，而对于切片来说，方括号内的表达式是在列表上下文中的。由这个特性可以引出另一个比较难解释的、貌似有些怪异的行为。

假设你想给数组@text 的尾部加上内容为 'EOF' 的一行字符串作为最后一个元素。你可以这么写：$text[@text] = 'EOF'。但如果写成 @text[@text]呢？那就犯了一个貌似很无辜、但却非常严重的错误：

```
chomp( @text = <STDIN> );
@text[@text] = 'EOF';
```

原因是 Perl 会把方括号中的数组 @text 看作是在列表上下文环境。在标量变量中，@text 返回的是其中元素的个数，但在列表上下文中，它返回的是数组元素本身。所以上面例子中方括号里的@text 会在列表上下文中返回一个切片，而这个切片恰恰就是数组 @text 中的每一行。

紧接着，每行内容都会被当作是下标转换成数字。如果每行都恰好只是纯文本（不像数字或者不是以数字开头的字串），那会全都转成数字 0，所以这个切片看起来就相当于 @text[0, 0, 0, 0, ...]。于是 EOF 会被赋值给这个切片中的第一个位置上的元素，而后续位置上的元素都会被赋值为 undef，又因为后续位置都指向数组中的第一个元素，所以该元素马上又会被不断重置为 undef，而其他位置上的数组元素不变。

够混乱吧？！

养成时刻检查类似 @a[0]这样的单个元素切片的习惯吧。一般来讲，单个元素的切片并不常见（尽管某些小技巧会用到），如果单个元素的切片出现在赋值操作左边，多半是理解上的失误。Perl::Critic 模块（见条款 112）可以帮你找出这类问题。

3. 方便而实用的切片

除了从某个列表结果中选择多个元素外，一般的 Perl 新手都不会刻意去使用切片：

```
my ( $uid, $gid ) = ( stat $file )[ 4, 5 ];

my $last = ( sort @list )[-1];

my $field_two = ( split /:/ )[1];
```

www.TopSage.com

但切片确实可以完成一些有趣（或奇特）的功能。比如说，下面的代码展示了使数组@list中第五个到第九个元素反转的两种不同方法：

```
@list[ 5 .. 9 ] = reverse @list[ 5 .. 9 ];
@list[ reverse 5 .. 9 ] = @list[ 5 .. 9 ];
```

切片这种方式还让两个元素的交换变得更简单：

```
@a[ $n, $m ] = @a[ $m, $n ];            # 交换$a[$m]和$a[$n]
@item{ 'old', 'new' } =
@item{ 'new', 'old' };                  # 交换$item{old}和$item{new}
```

4. 用切片给数组重新排序

在排序中也可用到切片（见条款22）。

给定两个元素互相对应的数组@uid和@name，下面的例子展示了根据@uid中元素的数字大小来对@name进行排序的功能。sort函数对@name的下标重新排序，然后将排序后的列表结果以切片的形式返回给数组@name：

```
@name =
  @name[ sort { $uid[$a] <=> $uid[$b] } 0 .. $#name ];
```

这段代码可以很好地完成任务，但你的同事要想理解这段代码，恐怕就得费点周折了。

5. 快速方便地创建散列

如果想以两个列表为数据源来创建散列，我们可以使用散列切片的方式。同样，我们也可以用散列切片的方式将一个散列的内容并到另一个散列中去，抑或是从一个散列中去除另一个散列所包含的相同内容。

创建以A到Z为键、1到26为对应值的一个包含26个元素的散列表其实很简单：

```
@char_num{ 'A' .. 'Z' } = 1 .. 26;
```

将一个新散列的元素并到一个已有的散列中也很简单（覆盖相同的元素，追加新增的元素）：

```
@old{ keys %new } = values %new;
```

后面的方式更简练，且效果相同，但效率没有前面的那种表达方式高：

```
%old = ( %old, %new );
```

将一个散列从另一个散列中去除，只要给定要删除键的列表，将对应键值对从散列中删掉就可以了：

```
delete @name{ keys %invalid };
```

上面一行代码明显比下面这段代码简洁：

```
foreach $key ( keys %invalid ) {
  delete $name{$key};
}
```

6. 要点

- 可以使用切片来选择列表、数组或是散列中的多个元素。
- 当只需要单个元素时，别用切片形式。
- 记住左值切片会让右边的表达式处于列表上下文环境。

www.TopSage.com

条款 12 理解上下文及其对操作的影响

Perl的创造者 Larry Wall 本是一名语言学家，因此 Perl 多少也有一些语言学上的特性。我们知道，两个人对谈话内容的理解是依赖于上下文的，**Perl 也依赖于操作符本身所提供的上下文来决定该如何处理数据**。所以，我们应该更多关注的是如何处理数据，而不是数据本身。

除了 Perl 操作符可以提供上下文环境，我们自己也可以实现上下文相关的代码（见条款 45）。

1. 数字和字符串上下文

在 Perl 里面，如何处理数据，如何求值表达式，全都取决于上下文。而有时候，上下文环境完全是由操作符来决定的。**算术操作符会将数据当成数字看待，字符串操作符会把数据当成字符串看待：**

```
my $result = '123' + '345';     # 值为数字 468

my $result = 123 . 345;         # 值为字符串 123456
```

记住，在做比较操作（见条款 6）时一定要用正确的操作符。如果把数字当成字符串来比较，得到的多半是个诡异的结果。就字符串比较而言，12 要比 2 小，这是因为字符串是逐个字符依次比较，而 1 要比 2 小：

```
if ( '12' lt '2' ) { print "12 is less than 2!\n"; }
```

类似地，用数值比较操作符来比较字符串，一样有可能得到莫名其妙的结果。就数值比较而言，字符串 foo 和 bar 都会被转换成数字 0 来比较（见条款 8）。因此从数值上来说，二者是相等的：

```
if ( 'foo' == 'bar' ) { print "Oh noes! foo is bar!\n" }
```

2. 标量和列表上下文

当操作需要的是单个对象时，Perl 会使用标量上下文。而当操作期望的是多个对象时，Perl 使用的就是列表上下文。

因为累加操作符（无论是操作数字还是字符串）两边需要的都是单个操作对象，所以它提供的是标量上下文：

```
1 + 2
'foo' . 'bar'
```

列表操作符期望操作的是多个元素。比如说 print 操作符，它会取得一列元素并输出：

```
print reverse( 1 .. 10 ), "\n";
```

根据定义，循环 while 的条件语句是标量上下文环境，而 foreach 的条件语句则是列表上下文环境：

```
while ( 标量上下文环境 ) { ... }
foreach ( 列表上下文环境 ) { ... }
```

这意味着行输入操作符用在它们两者中会有不同的行为。假设我们要遍历某个文件句柄，对

www.TopSage.com

其中的每行内容进行处理，一旦发现“__END__”标志就立即停止。在while循环中，我们会依次读入一行内容，进行处理，然后结束本次循环，继续读入文件句柄中的下一行：

```
while (<STDIN>) {
  last if /__END__/;
  ...;
}
```

```
my $next_line = <STDIN>;
```

如果改用 foreach 的话，可能就有问题了。由于它提供的是列表上下文环境，哪怕我们设定了提前退出循环的条件，它都会一次性将所有内容从 STDIN 中读出，而且此后 STDIN 句柄里就没有其他内容可读了：

```
foreach (<STDIN>) {
    last if /__END__/;
    ...;
}
```

```
my $next_line = <STDIN>; # 糟糕，没其他内容了！
```

一般而言，在实际用到之前就先读入的方式并不好。所以这种情况下还是用 while 循环比较合适。

3. 由赋值操作决定的上下文

赋值操作符也会提供上下文环境。它知道某次赋值操作是针对单个元素还是一列元素。如果是赋值给某个列表，那么赋值操作符会给它右边提供列表上下文环境。下面都是列表赋值：

```
my ($n)     = ( 'a', 'b', 'c' );     # $n为'a'

my ($n, $m)= ( 'a', 'b', 'c' );     # $n为'a', $m为'b'

my @array   = ( 'a', 'b', 'c' );

my @lines   = <STDIN>;               # 读入所有行
```

要注意的是，类似 my ($n) 这样的单个元素列表，它和包含任意个元素的列表本质没有什么区别。尽管只有一个标量在里面，但是标量两边的圆括号决定了它是列表操作。

给裸标量赋值（赋值操作左边没用圆括号括起来）指定的就是**标量上下文**（scalar context）。

```
my $single_line = <STDIN>;      # 只读入一行
```

这种情况有时候会比较让人迷惑，比如在下面的例子中$n的赋值会是什么呢？

```
my $n = ( 'a', 'b', 'c' );      # $n为'c'
```

为了理解上面这个看起来有些奇怪的结果，你需要了解列表和数组间的差异（见条款9）。

4. 空上下文环境

当我们调用某个子程序或使用某个操作符时，没有提供保存或使用其返回结果的目标时，就是**空上下文**（void context）：

www.TopSage.com

```
some_sub(@args);

grep { /foo/ } @array;

1 + 2;
```

某些情况下，Perl 会告诉我们这种操作毫无意义。比如上面的 grep 操作，既然不需要保存搜索结果，那还费力找个啥？

碰到类似 1+2 这样的情况，如果打开了 warnings 开关（见条款 99），Perl 会给我们发出一个警告信息：

```
Useless use of a variable in void context
```

5. 要点

- Perl 会依据你的操作自动判断当前的上下文环境。
- Perl 会根据数字或字符串上下文来对标量作相应解释。
- 在列表上下文和标量上下文中的计算结果可能完全不同。

条款 13　用数组或散列集结一组数据

数组和散列是 Perl 中内置的两种数据类型，它们都属于集合（collection）类型，主要是用来集结一组相关的数据。

1. 不要使用带数字编号的变量

有些新手知道标量但不知道数组，我们经常会在他们所写的代码中看到用多个独立标量保存一系列值：

```
my $fib_0 = 1;
my $fib_1 = 1;
my $fib_2 = 2;
my $fib_3 = 3;
my $fib_4 = 5;
```

而有些稍微懂得多一点的程序员虽然知道用数组，但方法却一样笨拙：

```
$fib[0] = 1;
$fib[1] = 1;
$fib[2] = 2;
$fib[3] = 3;
$fib[4] = 5;
```

其实将值保存到数组里面的操作可以非常简单直观：

```
my @x = ( 1, 1, 2, 3, 5 );
```

如果用 qw() 操作符的话，还可以更轻松些：

```
my @x = qw( 1 1 2 3 5 );
```

让我们举个更复杂的例子，例如，保存一个二维坐标，最原始的办法是列出两组独立的变量：

www.TopSage.com

大家网 TopSage.com

```
my $x0 = 1;
my $y0 = 1;
my $x1 = 2;
my $y1 = 4;
```

但这个方法远不如保存到数组方便，为了将坐标点按组分在一起，我们得以**元组**（tuple）的方式保存，也就是将每组坐标点分别保存在代表它们的数组引用里：

```
my @points = ( [ 1, 1 ], [ 2, 4 ] );
```

要实现遍历所有坐标点也很简单：

```
foreach my $point (@points) {
  print "x: $point->[0] y: $point->[1]\n";
}
```

2. 避免使用成堆变量

另一个新手常见的问题是，将一组逻辑上相关的信息保存到多个不同的变量中。比如和某个人相关的三个数据，新手可能会创建三个独立的标量变量来保存：

```
my $person_name = 'George';
my $person_id   = '3';
my $person_age  = 29;
```

尽管用了三个变量，但这些数据实际都是联系到一个人身上的，而且我们可以发现，这三个变量的名字彼此之间也有联系，前缀都是 person。每当需要用到这些数据时，初学者不得不把这三个值依次传入，还得保证参数传递时的顺序内外一致：

```
some_sub( $person_name, $person_id, $person_age );
```

如果再遇到上面这种模式，我们完全可以把它们转换成散列。除去各变量名的前缀 person，作为散列的键来定义：

```
my %person = (
  id => 3,
  name => 'George',
  age => 29,
);
```

现在我们就不用担心怎么传递 George 的信息了，无论何时都可以把这个散列的引用作为参数进行传递：

```
some_sub( \%person );
```

如果有好几个人，可以分别定义不同的散列，以数组的方式集结：

```
my @persons = ( \%person1, \%person2 );
```

初学者刚开始可能会经常像上面这么写，每次创建散列表都使用具名散列。而作为数组元素的散列引用，并不一定都要有名字，我们完全可以创建匿名散列引用：

```
my @persons = ( { id => 1, ... }, { id => 2, ... }, ... );
```

3. 要点

- 避免用一组名字相近的标量变量保存分组数据。

www.TopSage.com

- 用集合型的数据类型保存分组数据。
- 用散列表示一个实体对象，利用键值表示数据属性。

条款 14　用 bignum 处理大数

尽管 Perl 会尽力向用户隐藏内部实现细节，但在某些情况下，比如说它能展示的最大（或最小）数字，还是得暴露给用户。对于还很常见的 32 位的 perl 来说，它的整数范围被限定在 32 位，试试看下面这条单行程序：

```
% perl -le 'print 12345678901234567890123345';
```

我们会得到以指数形式表示的结果，这大概和你预计的不同：

```
1.23456789012346e+24
```

还可以试试这段计算阶乘的代码：

```
my $factorial = 1;
foreach my $num ( 1 .. $ARGV[0] ) {
  $factorial *= $num;
}

print "$factorial\n";
```

如果在 32 位的 perl 中，计算到 18 的阶乘时，结果就会变成指数形式：

```
% perl factorial.pl 17
355687428096000
% perl factorial.pl 18
6.402373705728e+15
```

尽管是指数形式，数值上还是完全正确的，但计算到 21 的阶乘时（51 090 942 171 709 440 000），perl 就开始丢失精度了：

```
% perl factorial.pl 21
5.10909421717094e+19
```

1. 确保数字精度

要解决 perl 丢失精度的问题，我们可以启用 bignum 编译指令：

```
use bignum;

my $factorial = 1;
foreach my $num ( 1 .. $ARGV[0] ) {
  $factorial *= $num;
}
print "$factorial\n";
```

现在，我们的程序可以处理更大的数字了：

```
% perl factorial.pl 100
933262154439441526816992388562667004907159682643816210
```

www.TopSage.com

```
46859296389521759999322991560894146397615651828625369
792082722375825118521091686400000000000000000000000
```

如果系统资源允许的话，我们现在可以处理任意大小的数字。通过正确选择运算对象的类型及在适当时候进行类型转换，不管是整数还是浮点数，都可以自动处理。由于底层系统对数据类型本身大小有所限制，使用任意大的数字会让程序运行耗费更多的硬件资源和时间，没办法，这就是我们追求精确结果的必然代价。

2. 限制 bignum 的作用范围

一旦启用了 bignum 编译指令，所有的数字和数学运算都会使用大数形式。你可能不想这样，因为 bignum 运算每次都会将数字转换成 Math::BigInt 或 Math::BigFloat 对象。这会极大地影响性能，因为所有的计算都变成了对象方法调用。

如果程序中的大部分地方都需要用到 bignum，那么小范围内不用的地方可以用 no bignum 指令临时关闭：

```
{
    no bignum;

    # 这里的数字就是普通类型的了
    my $sum = $n + $m;
}
```

同样的道理，如果我们只想在程序的小范围内使用 bignum，也可以局部开启：

```
{
    use bignum;

    # 这里的数字就是大数类型的了
    my $sum = $n + $m;
}
```

如果只有某些特定数据需要使用 bignum，那么可以通过构建对象的方式表示大数。对象的操作方法一如既往，而对于大数的特别处理也仅仅发生在这些对象上：

```
use Math::BigInt;

my $big_factorial = Math::BigInt->new(1);
foreach my $num ( 1 .. $ARGV[0] ) {
  $big_factorial *= $num;
}

print "$big_factorial\n";
```

这时，只有我们自己创建的大数对象是 bignum 类型，其他数字仍旧保持内置类型（及大小）不变。

3. 要点

- Perl 内置的数字精度受限于本地操作系统的内部架构。
- bignum 编译指令可以让你使用任意精度的数字。
- 有选择地使用 bignum，以免程序运行过于迟缓。

www.TopSage.com

第2章 Perl的地道用法

Perl语言的设计者是一位语言学家，所以和任何一种人类语言相同，Perl也有很多习语。

这里所说的习语，一部分是指那些非常精炼的Perl写法，另一部分是指约定俗成的写法。经验老道的Perl程序员常常会这么写，新手也往往乐此不疲。

具体来说，到底哪些是习语哪些是惯例，并没有明确的界限，判断标准也因人而异。不管是简单的还是复杂的算法，Perl都有多种表达处理方式，而其中有些明显要比其他方法更“正确”些。尽管Perl有句口号是“条条大路通罗马”，但其实往往“大多数路都是错的”，或者说，“总有某些路要更好一些”。

习语和惯例对Perl来说更为重要，相比之下，C、Bourne shell或者C shell就不那么依赖它。这就是为什么我们说C语言编程其实并不算复杂的原因之一。你要是觉得这么说太疯狂，请去任何一个计算机书店看看就会明白。书架上不少的书会告诉你如何灵活使用C语言，不过大都没有C++的书厚。同样的，尽管有很多关于shell编程的书，但在有关Perl的书对比下都会显得薄些。

Perl是一门富于表达能力的语言，所以也往往非常简练。Larry Wall在设计这个语言时，特别注重代码的精炼。这就是我们常常听到的**赫夫曼编码**（Huffman coding）原则：最常用的那些操作应该只需要最少次数的键盘敲击。这在Perl的高级特性中非常常见，比如<>、正则表达式以及grep的用法中都能看到：

```
# 对换 $a 和 $b
( $a, $b ) = ( $b, $a );

# 从文件或标准输入读取各行，排序后打印
print sort <>;

# 打印所有带有 joebloe 这个词的行
print grep /\bjoebloe\b/, <>;

# 把 @n 中可以被 5 整除的数筛选出来，存入@div5

my @div5 = grep { not $_ % 5 } @n;

# 把"123.234.0.1"转化成二进制整数 0xb7ae0010 的方法之一
$bin_addr = pack 'C4', split /\./, $str_addr;
```

你可以用其他方法完成同样的任务，不过要是用其他语言编写，多半写出来的是更长更低效

www.TopSage.com

的代码。比如像<>的功能，虽然也能用你自己的方式表达出来，但结果一定繁杂而冗长，最终导致程序中真正有用的代码变得很“模糊”。而调试和维护也会变得更为麻烦，因为程序的长度和复杂度都已经大大增加。

从某种意义上来说，习语和编程风格会有交集。虽然像 `print sort <>` 这样的代码无可挑剔，但总有处于灰色地带的情况：

```
# 逐行打印%h 中的所有键值对
foreach my $key ( sort keys %h ) {
  print "$key: $h{$key}\n";
}

# 另一种打印键值对的方法
print map "$_: $h{$_}\n", sort keys %h;
```

第一段是非常直白的写法，不但高效，而且可读性强。因为它只使用了语言的基础特性。而第二段代码更短，而且对某些人来说，它显得“酷毙”了，因为这里用了 map 操作符和列表上下文来代替普通的 foreach 循环。无论如何，在写出这样的代码之前，请先考虑自己或其他人将来读到这段代码时的感受。这种写法不但有些难懂（但也不能算非常难懂），执行效率也有可能变得低下。

每个 Perl 程序员都应该掌握基础习语，同时也该对更高级的习语有所了解。作为程序员，应该熟练并习惯于使用那些高效、简洁并且可读性好的习语。而那些高度复杂的习语并不那么重要，具体使用要看场合，主要取决于程序作者、读者身份以及程序本身等各方面因素。

本章我们会展示许多 Perl 习语，其实有些你肯定已经看到过。相信你会立刻喜欢并着手使用那些简单的习语。而对于那些稍显复杂的，请自行决定哪些合用，哪些不合用。

Perl 代码反应了作者的品性。只要你喜欢，平铺直叙地写代码也毫无问题。编程就像盖房子，是一门堆砌砖块的艺术。虽然简单，但只要能顺利运行，那就很好。只不过循规蹈矩出来的作品，大抵不会有什么特色。

而与此相反，你可能一心想着要把所有干净利落的语法特性都用起来。这就好像成天蹲在工具房浪费时间琢磨该用哪种新式强悍工具的建筑师，忘了完成任务才是工作的根本。能以各式最新最酷的编码技术当然很好，但往往你最终会发现，最朴素最经得住考验的，还是那些普通的锤子。

或许经过一段时间后，你能在上述两者之间取得某种平衡。

有时需要的是 `s/\G0/ /g`，而有时需要的，仅仅是一句 `$a = $b`。

条款15　为优雅、简洁而使用$_

Dollar underscore 或者写为 $_——你可能喜欢它，也可能厌恶它，但不管怎样，如果想要成为一名熟练的 Perl 程序员，就必须要深刻理解它。

$_ 是许多操作符的默认参数，有时也是一些控制结构语句的默认参数。请看下面示例：

```
# $_作为默认参数
print $_;   # 写明 print 的默认参数是 $_，实际可以省去
print;      # 省略后效果完全相同
```

www.TopSage.com

TopSage.com


```
print "found it"
  if $_ =~ /Rosebud/; # 用于匹配和替换操作
print "found it"
  if /Rosebud/;       # 效果同上
$mod_time = -M $_;    # 大多用于文件句柄测试
$mod_time = -M;       # 效果同上

foreach $_ (@list) { do_something($_) } # 用于 foreach 循环
foreach (@list)    { do_something($_) } # 效果同上

while ( defined( $_ = <STDIN> ) ) {
  # while 是一种特殊情况
  print $_;
}
while (<STDIN>) { print } # 效果同上
```

最后一个例子在条款 7 中也曾涉及，展现的是它的特殊用法，在 while 循环中使用 <filehandle> 操作符是一种简写，实际的操作过程是：逐行读入文件到 $_，直到文件末尾。它会自动检查文件读取的返回值，一旦到达文件末尾，会返回 undef 宣告结束。

看上去有些凌乱，不过碰到困惑，最好的办法还是查阅 Perl 文档指南。不必记住整个文档，因为它是用来查阅的。比如说，split 函数会对 $_ 作何特殊处理？查查文档就好。

其实 $_ 也可以算是一个普通的标量变量，可以用来存放数据、打印内容、修改内容等。不过它还有些独特之处需要你特别注意。

1. $_和 main 包

在 Perl 5.10 之前，$_ 只属于 main 包，哪怕当时正在执行的代码位于其他包内也是如此：

```
package foo;
$_ = "OK\n"; # 这仍然等同于$main::_

package main;
print;       # 这会打印出"OK"
```

其实，所有的特殊变量（[$@%] 中的任一个加上标点符号，以及某些其他变量）都是这样的。你确实可以用 $foo::_ 这样的形式，但它只是一个普通变量，不会具有和 $_ 一样或类似的特殊功能。

2. $_的本地化

在 Perl 5.10 之前，你只能用 local 来对 $_ 进行本地化。使用特殊变量时，多半都需要限制它的生命期和作用域。有了 local 之后，你可以在当前范围内使用自己的 $_，直到所在作用域结束：

```
{
  local $_;

  # ... 使用自己的 $_ 版本

  some_sub(); # 也会使用自己的 $_
}
```

www.TopSage.com

有了 local，你可以在限定的作用域内随意修改特殊变量，包括在所调用的子程序中仍然可见和有效。如此一来，就可以在整个程序中使用自己的 $_ 版本。

Perl 5.10 版本以后，$_ 开始支持词法域，对它所作的修改，仅限于当下代码的作用域可见，从外部是无法访问的。即便是同一个作用域内所调用的子程序 some_sub，也不能访问到词域类型的$_变量：

```
{
  my $_;

  #... 使用自己的$_版本

  some_sub(); # 它看不到你的 $_
}
```

3. 编程风格与 $_

在使用 $_ 的地方，往往 $_ 都隐匿于无形。像下面这段代码，你能计算出总共隐形地用到了几次 $_ 吗？

```
while (<>) { # 第一次
  foreach (split) { # 第二、三次
    $w5++ if /^\w{5}$/ # 第四次
  }
}
# 查找以.txt 结尾，并小于 5000 字节的文件
@small_txt =
  grep { /\.txt$/ and (-s) < 5000 } @files; # 第五、六次
```

正因为如此，有些 Perl 程序员会觉得 $_ 好像更容易让人迷惑，而不是化繁为简。甚至有一本书[①]中直白地指出，“很多 Perl 程序员在他们的程序中随意使用 $_，相比自己命名的变量，这样的代码更难读懂、更容易出问题。”真的是这样吗？看看下面哪种写法更容易读懂呢？

```
while ( defined( $line = <STDIN> ) ) {
  print $line if $line =~ /Perl/;
}

while (<STDIN>) { print if /Perl/ }
```

其实这里使用 $_ 并不会带来什么理解上的困惑。只要学一点 Perl 基础并善于查阅文档，那么所有的困惑都是暂时的。

4. 要点

- 多数操作符都使用 $_ 变量作为默认参数。
- 链接多项操作时，应避免使用临时变量，而改用默认变量。
- 使用$_前可通过本地化避免影响其他代码。

① David Till 所著的 *Teach Yourself Perl 5 in 21 Days*（Berkeley，CA：Sams Publishing，1996）。

www.TopSage.com

条款 16　了解其他默认参数

$_ 并非 Perl 里面唯一的用作默认参数的特殊变量，除此以外还有其他几个类似的。但和$_一样，不管哪个默认参数，有关的详细信息都在官方文档中，阅读也好，参考也罢，核心文档总归是最好的学习指南。

1. @_作为默认参数

在子程序中，shift 操作会默认用 @_ 作为参数。所以如果看到没有任何参数的这类调用，其实是在获取 @_ 最左边的元素：

```
sub foo {
  my $x = shift;
  ...;
}
```

这里要介绍一个有趣的语法，它能直接对被引用的数组进行传递，而且只有一行代码：

```
bar( \@bletch );

sub bar {
  my @a = @{shift};
  ...;
}
```

在这个例子中，Perl 会把 @{shift} 中的 shift 理解为变量名，而非内建函数。只要打开 strict 编译指令（见条款 3），你就会看到 Perl 实际上是这样理解的：

```
Global symbol @shift requires explicit package name ...
```

所以，你必须在花括号中加点什么，好让 Perl 知道这个标识符其实并非变量名，而是一个函数。一般最常见的做法就是添加圆括号：

```
my @a = @{ shift() };
```

也可以使用单目操作符规避歧义，比如用 +，实际的作用相当于占位符，让 Perl 知道 shift 并不是字符串：

```
my @a = @{ +shift };
```

要是不太适应这种将参数引用解开来的写法，当然也可以拆解成平常的几步来做，这是你的自由。

2. @ARGV 作为默认参数

在子程序之外，shift 会把 @ARGV 作为默认参数。知道了这点，我们就可以对命令行参数作自定义处理。比如把所有以连字符开头的参数当作开关选项，其他参数当作文件名：

```
foreach (shift) {
  if (/^-(.*)/) {
    process_option($1);
  }
  else {
    process_file($_);
```

www.TopSage.com

```
    }
}
```

当然，这项工作没必要自己来，Perl 有很多现成的模块用于处理程序参数，比如 Getopt::Long 模块。

另外请注意，shift 操作符总是会使用 @ARGV 或 main::@_，就算当前运行在其他包的名字空间中也一样。

3. 其他使用$_的函数

另外还有一些内建函数会默认使用 $_：-X 文件测试操作符(但 -t 除外)、abs、alarm、chomp、chop、chr、chroot、cos、defined、eval、exp、glob、hex、int、lc、lcfirst、log、lstat、oct、ord、pos、print、quotemeta、readlink、ref、require、reverse（仅限于标量上下文）、rmdir、say、sin、split、sqrt、stat、study、uc、ucfirst，以及 unlink。

4. 默认使用 STDIN

多数文件测试操作符都以 $_ 作为默认参数，但 -t 却以 STDIN 文件句柄作为默认参数。-t 使用 UNIX 的 isatty() 函数检查文件句柄是否可以交互。交互的意思，就是说终端前是否有一个人在敲键盘输入，或者其他自动回馈信息的机制。下面两条-t 调用是完全等效的：

```
print "You're alive!" if -t STDIN;
print "You're alive!" if -t;
```

使用 -t 操作符可以根据环境的交互能力决定程序行为。比如通过 -t 测试决定是否让命令行启动的 CGI 程序进入调试模式。类似的任务也可以用 IO::Interactive 模块实现，而该模块还能应付许多其他特殊情况。

从特定文件句柄读入单个字符的 getc 函数，也默认使用 STDIN 作为参数：

```
my $char = getc STDIN;
my $char = getc;
```

5. 其他默认参数

无需赘言，请参考表 2-1。该表总结了 Perl 内置函数所使用的其他默认参数。

6. 要点

- 并非所有操作符都将 $_ 作为默认参数。
- 有些函数对默认参数的使用是随上下文而变的，比如 shift。
- 如果不想使用默认参数，当然也可以自己指定要使用的参数。

表2-1　不使用$_作为默认参数的Perl内建函数

Perl 内置函数	默认参数
chdir	$ENV{HOME}、$ENV{LOGDIR}或$ENV{SYS$LOGIN}
close	默认文件句柄
dump	程序的开始
eof	最近读取的文件或者ARGV的最后一个文件

www.TopSage.com

（续）

Perl 内置函数	默认参数
exit	0
getpgrp	$$
gmtime	time函数
localtime	time函数
open	$FILEHANDLE
pop	子程序中使用@_，子程序外使用@ARGV
rand	1
reverse	仅在标量上下文中为$_
select	当前文件句柄
shift	子程序中使用@_，子程序外使用@ARGV
tell	最近一次读取的文件
write	当前文件句柄

条款17　常见简写和双关语

Perl这个语言的语法是上下文相关的，这有点类似人类语言。你可以充分利用这一点，在某些可以省略的位置上，让解释器自己找到默认要处理的东西，例如默认参数、$_、可写可不写的标点符号等。Perl通常都能根据上下文充分领会你的意图。

另外Perl也属于那种极富表达力的语言，但有时多变的语法并不能完美兼顾。所以某些情况下，你得亲自给它一点暗示，才能帮助它作出正确选择。接下来我们会给出一些建议，并指出需要小心处理的情况。

1. 使用列表赋值来进行变量对调

Perl并没有专门用于变量值对换的操作符，但你可以用列表赋值来完成这个任务。Perl会先计算等号右边的表达式，然后按对应位置赋值：

```
( $b, $a ) = ( $a, $b ); # 对换$a和$b的值

( $c, $a, $b ) = ( $a, $b, $c ); # 轮转$a、$b和$c的值
```

数组切片能让你用简单的语法随意置换数组内容：

```
@a[ 1, 3, 5 ] = @a[ 5, 3, 1 ]; # 对换个别数组元素
```

也可以进一步使用数组切片来完成数组奇偶元素的对调：

```
@a[ map { $_ * 2 + 1, $_ * 2 } 0 .. ( $#a / 2 ) ] = @a;
```

2. 用[]或()[]转为列表上下文

有时需要强制Perl在列表上下文计算某个表达式。比如用正则表达式切分字符串，开始的时候你可能会这么写：

```
# 按+号切分$_中:之前的部分
my ($str) = /([^:]*)/;
my @words = split /\+/, $str;
```

www.TopSage.com

为了避免使用临时变量$str，可以把上面两条语句合并为一条。不过这得需要一些技巧，因为第一个模式匹配在标量上下文（也就是split期望处理的目标字串）的地方不会返回匹配到的字串。我们可以利用切片，将它转为列表上下文，然后取第一个位置上的值：

```
my @words = split /\+/, (/([^:]*)/)[0];
```

如果想要一步完成列表计算结果到引用的转换，可以用匿名数组构造符[]。引用操作符\施加在列表上，只会返回一个新的列表，包含的是原来列表元素的引用，而非数组引用（见条款61）：

```
my $wordlist_ref = \( split /\++/, $str );  # 错误

my $wordlist_ref = [ split /\++/, $str ];   # 正确
```

3. 用=>构造键值对

大箭头操作符=>的功能其实与逗号操作符差不多，唯一的差别在于：如果=>左边的标识符能识别为一个单词，那么Perl就会自动将它当成一个字符串，而不是函数调用。这样一来你就能在=>的左边使用time这样的字串：

```
my @a = ( time => 'flies' ); # 这里time只是一个单词
print "@a\n";                # 打印"time flies"

my @b = ( time, 'flies' );   # 这里的time是内建函数
print "@b\n";                # 打印"862891055 flies"
```

使用=>还能让数据的初始设定式更加美观，特别是对于散列结构的初始设定式来说更是如此。哪一列是键，哪一列是值，一目了然：

```
my %elements = (
  'Ag' => 47,
  'Au' => 79,
  'Pt' => 78,
);
```

在表示散列键的地方可以省略引号：

```
my %elements = (
  Ag => 47,
  Au => 79,
  Pt => 78,
);
```

有时连=>都可以省略。如果键和值都是没有空格的字串，就可以用qw操作符（见条款21）直接构造：

```
my %elements = qw(
  Ag  47
  Au  79
  Pt  78
);
```

4. 用=>操作符来模拟命名参数

你可以为函数调用模拟命名参数。只要在子程序内作好设置，就可以让它以散列的形式接受

参数。而要快速创建散列，只需把默认值放在前面，后面加上函数得到的参数即可。在后面的例子中，%params 预先包含了参数 align 的默认取值：

```
sub img {
  my %params = ( align => 'middle', @_ );

  # 把散列的键和值输出成 HTML 标签的格式
  print "<img ",
    (
    join ' ', map { "$_=\"$param{$_}\"" }
      keys %param
    ),
    ">";
}
```

在调用 img 子程序时，可以用自己的键值对作为参数：

```
img( src => 'icon.gif', align => 'top' );
```

这里设置的 align => 'top'覆盖了 %params 里的默认取值，所以最终你会得到正确的 img 标签：

```
<img src="icon.gif" align="top" />
```

请不要在实际应用中照搬上面的做法，这里只是演示概念。处理 HTML 文件还请使用专门的 HTML 模块。

5. 用 => 指明操作方向

最后还有一个关于 => 操作符的有趣用法，就是作为“语法糖”来指明操作方向。比如在 rename 函数中，用它来表示从旧文件名改为新文件名：

```
rename "$file.c" => "$file.c.old";
```

请不要混淆 => 和 ->，它们的意义不同，后者是用来对引用取数据的（见条款 58），或用于面向对象的方法调用。

6. 小心使用 {}

圆括号、方括号、尖括号和花括号在 Perl 里面都是多面手。Perl 在处理花括号（或者其他括号）时会先看它所处的上下文，然后再决定该如何解释。一般得到的结果都符合常理，但也有让人意外的情况。

使用花括号时请特别小心，它大概算是 Perl 里面功能最多的标点符号了。花括号可用来圈定程序块，用作变量名分隔符，创建匿名散列，或者进行散列元素的存取、解引用，等等。面对这么多用途，如果太过纠结的话肯定会晕头转向。不过好在解释器能清楚地区分匿名散列构造和程序块定义。

如果看到花括号中有个孤零零的加号，又貌似毫无必要，那它很可能是被有意放在那里的。Perl 的单目加号对跟在后面的参数没有实际影响，但可借此对语法解析进行修正。比如对某个返回数组引用的函数进行解引用操作时，将函数名放在 {} 内，会被当成是软引用（可以用 strict 捕获，见条款 3）：

www.TopSage.com

```
my @a = @{func_returning_aryref}; # 错误!
```

而以下三种写法都能正常工作，因为你给了Perl提示：func_returning_aryref是一个函数名：

```
my @a = @{ func_returning_aryref() }; # 正确，因为有圆括号

my @a = @{ &func_returning_aryref };  # 正确，因为有&符号

my @a = @{ +func_returning_aryref };  # 正确，因为有加号
```

若是不巧，可能会遇上难以分辨匿名散列还是程序块定义的窘境。像后面这段代码，假设我们想要函数返回的键值对列表，以匿名散列的形式从eval代码块中返回。花括号是Perl里面被重载定义得最多的一个操作符，所以出现歧义的可能性比较大。碰到实在没有可供分辨的线索，我们可以手工提供一个，好让Perl排除歧义。下面的写法是将匿名散列作为返回表达式的一部分，这样Perl就知道函数key_value_pairs()返回的内容，被用于构造匿名散列表：

```
my $hashref = eval {
  return { key_value_pairs() } # 正确
}
```

或者也可以用单目加号来暗示Perl，后面的花括号并非块定义，那就只可能是匿名散列构造：

```
my $hashref = eval {
  +{ key_value_pairs() } # 正确
}
```

但若没有以上提示的话，Perl会猜测花括号是用来为key_value_pairs()限定作用域的，而这个猜测显然不是我们的本意：

```
my $hashref = eval {
  { key_value_pairs() } # 可能会导致问题
}
```

最后还要提一点，Perl会把所有在花括号中的单个标识符（包括两边的空白）都理解成字符串。所以如果此处是一个函数名字的话，实际并不会调用函数，而仅仅是将此名字当作普通字串处理。为排除歧义，只需破坏“单词”的形式，比如借助加号：

```
${shift} = 10;     # 完全等同于$shift = 10

sub soft { ${ +shift } = 10; } # 是软引用!
soft 'a';                      # 相当于$a = 10
```

鉴于这种写法很容易让人困惑，请不要这样设置变量。如果看到别人这么用，最好劝他改掉。

7. 用@{[]}或eval {}的形式来复制列表

有时候我们希望操作的是某个列表的副本，以免原始数据遭到破坏。比如查找缺失的.h头文件，可以先读取所有的.c文件，然后将文件名换成.h结尾，再按此列表核对哪些文件不存在：

```
my @cfiles_copy = @cfiles;
my @missing_h =
  grep { s/\.c$/\.h/ and not -e } @cfiles_copy;
```

www.TopSage.com

Perl 里面没有提供复制数据结构的函数，如果需要一份列表的匿名副本，可以把列表放在匿名列表构造操作符[] 中，然后对它进行解引用，以符合 grep 函数对列表上下文的预期：

```
my @missing_h =
   grep { s/\.c$/\.h/ and !-e } @{ [@cfiles] };
```

另一种产生临时副本的方法是把它放在 eval 块中，它能返回块内部最后一个表达式的计算结果：

```
my @missing_h =
   grep { s/\.c$/\.h/ and !-e } eval { @cfiles };
```

遇到这种情形，请尽量使用 eval 方式，比起前一种写法它更高效。

不过，你还是应该先考虑一下是否真的需要临时副本。在前面这个例子中，完全可以在 grep 里面作一些改进，引入一个 $_ 变量的临时副本就可以了：

```
my @missing_h = grep {
  my $h = $_;
  $h =~ s/\.c$/\.h/ and !( -e $h )
} @cfiles;
```

另外请注意，以上手法创建的都是“影子副本”即完整独立的副本。如果原始列表中的元素是引用的话，复制的结果就会和原始数据共享。如果确实需要一份完全独立的深层副本，请使用 Storable 模块的 dclone 函数：

```
use Storable qw(dclone);

my $copy_ref = dclone( \@array );
```

现在的 $copy_ref 就和原来的 @array 完全不相干了。

究竟使用哪种技术，取决于实际要解决的问题。但不管怎样，对列表的复制往往会给人造成困扰，使用的时候要注意区分。

8. 要点

- Perl 语法渊源广博，因此颇多双关。
- 用 => 能更好地展示数据间的关系。
- 可以用 dclone 对数组进行深层次地复制。

条款 18　避免过分依赖标点符号

Perl 程序容易变成满是标点符号的样子，有时甚至会达到令人畏惧的地步。标点符号的滥用会造成程序可读性下降，聪明的程序员会想尽办法，充分利用 Perl 本身的特性避免使用不必要的标点符号。

1. 无括号方式调用子程序

调用子程序时，前置的 & 符号、函数名后面的圆括号都是可写可不写的，组合起来便有好几种形式。下面这些写法，只有最后一种需要在运行之前先作函数声明或函数定义：

www.TopSage.com

```
&myfunc( 1, 2, 3 );
myfunc( 1, 2, 3 );

myfunc 1, 2, 3;
```

传统的&语法也有它的用处：如果子程序的名字恰好是保留的关键字，那么“**&子程序名**”就是唯一的调用方法，例如&for。而类似列表操作（也就是不带&符号和圆括号）的语法则要求事先明确函数定义或声明。这种写法是最简洁的，但有个细节需要注意。

如果Perl的解释器还没处理到有关子程序的定义，就不能用列表操作语法调用：

```
myfunc 1, 2, 3; # 错误
sub myfunc { }
```

此外，这里按照的是源代码中的先后顺序，所以放在BEGIN块里面的函数定义实际上是在函数调用后面：

```
myfunc 1, 2, 3; # 错误

BEGIN {
  sub myfunc { }
}
```

另外，子程序必须在编译前就已完成定义，所以通过eval来动态构造函数的方法也不能用列表语法调用：

```
eval "sub myfunc {}";
myfunc 1, 2, 3; # 错误
```

BEGIN块可以写在子程序调用前，但看上去会比较怪异：

```
BEGIN { eval "sub myfunc {}" } # 可以工作，但很怪异
myfunc 1, 2, 3;
```

如果能提前声明子程序，就算推后定义，也一样可以使用列表语法。因为Perl已经知道myfunc是一个子程序，那它在解析源代码时就不会困惑：

```
use subs qw(myfunc);

myfunc 1, 2, 3;
sub myfunc { }
```

2. 用and和or替代&&和||

另一个可以借助的功能是优先级较低的逻辑短路操作符and和or。在操作符的结合优先级表中，它们位于最低一级（另外同一级别的还有not和xor操作符，但都没这两个常用）。通过逻辑短路，可以免去对表达式的分组，有效减少圆括号的使用。

最低优先级的and和or操作符允许我们省略列表操作、赋值操作和匹配操作中使用的圆括号。对照各组示例前后，哪一种写法看起来更好？

```
print("hello, ") && print "goodbye.";
print "hello, " and print "goodbye.";

( my $size = -s $file )|| die "$file has zero size.\n";
```

www.TopSage.com

TopSage.com


```
my $size = -s $file or die "$file has zero size.\n";
( my $word =~ /magic/ )|| $mode = 'peon';
my $word =~ /magic/ or $mode = 'peon';

open( my ($fh), '>', $file )
  || die "Could not open $file: $!";
open my ($fh), '>', $file
  or die "Could not open $file: $!";
```

请记住，花括号中的最后一个分号总可以省略。这一点对那些单语句块来说非常有用，尤其是 map、grep、do、eval 这样的操作符：

```
my @caps = map { uc $_; } @words;   # 这里的分号意义不大
my @caps = map { uc $_ } @words;    # 看起来更利索些
```

最后一种避免使用圆括号和花括号的方法是使用表达式修辞，即“后向条件式”语法。一旦习惯，你就会爱上这种简练的写法：

```
if (/^__END__$/) { last }  # 实在平淡无奇

last if /^__END__$/;       # 这样更自然，不是吗？
```

3. 要点

- 避免滥用圆括号和花括号。
- 尽可能采用列表语法调用子程序。
- 借助低优先级的 and 和 or 减少无谓的括号。

条款 19　调整列表格式以便于维护

尽管之前我们说要避免标点符号的无谓使用（见条款 18），但凡事无绝对，有些情况下额外添加标点符号反而利大于弊。作为 Perl 的设计者，Larry Wall 认识到包容其他语言中看似错误的语法也很必要。如果有人一而再，再而三地犯同一个错误，那么最好的解决办法是把错误包容起来，不去理会。

所以，Perl 允许你在列表最后一个元素末尾加上一个逗号：

```
my @cats = ( 'Buster Bean', 'Mimi', ); # 不会出错
```

那么以后再加入新条目时，就不必照顾前面的代码补上逗号：

```
my @cats = ( 'Buster Bean', 'Mimi', 'Roscoe' );
```

如果按每行一个元素的格式排布，看起来会更清楚：

```
my @cats = (
  'Buster Bean',
  'Mimi',
  'Roscoe',
);
```

这样，新增或删除某个元素时，都不必担心末尾的逗号影响列表结构。临时注释掉某个元素也很方便：

www.TopSage.com

```
my @cats = (
  'Buster Bean',
  # 'Mimi',
  'Roscoe',
);
```

对散列表来说，这种排布格式的优势更为明显，每行写一个键值对，按列对齐：

```
my %spacers = (
  husband  => "George",
  wife     => "Jane",
  daughter => "Judy",
  son      => "Elroy",
);
```

如果列表元素都不含空格，可以直接使用 qw 操作符初始化，这样逗号和引号可以一并省略：

```
my %spacers = qw(
  husband   George
  wife      Jane
  daughter  Judy
  son       Elroy
);
```

要点

- 列表元素独立成行，条理清晰，便于编辑。
- 习惯在列表末尾附加逗号，添补新元素时就不会忘记补漏。
- 将散列键值对按列对齐，以便于维护。

条款20　善用 foreach、map 和 grep

Perl提供了好几种遍历列表元素的方法。

Perl程序员普遍趋向于避免使用C语言风格的 for 循环和按下标遍历列表的方式。使用下标的循环通常比其他循环方式慢，因为下标检索需要耗费 Perl 一定的计算时间，而 C 语言风格的代码结构复杂，叫人望而却步：

```
for ( my $i = 0 ; $i <= @array ; $i++ ) {
  print "I saw $array[$i]\n";
}
```

大多数程序员都爱用 foreach、map 和 grep 形式的循环。这三种方式有相似之处，不过各自的应用场合不同。如果要较真，任何一种形式的循环都可以用其他形式等价表示，但不合时宜的使用往往会使人困惑，困惑的不仅是那些将来会阅读你的代码的其他人，还包括你自己。所以，知才善用，才是最恰当的做法。

1. 通过 foreach 来进行列表元素的只读遍历

如果仅仅是要遍历列表中的所有元素，那么 foreach 循环就已足够：

```
foreach my $cost (@cost) {
  $total += $cost;
```

www.TopSage.com

```
}

foreach my $file ( glob '*' ) {
  print "$file\n" if -T $file;
}
```

请记住，如果没有特别设置的话，foreach 默认使用 $_ 作为控制变量：

```
foreach ( 1 .. 10 ) { # 打印一到十的平方
  print "$_: ", $_ * $_, "\n";
}
```

另外，所有用到 foreach 的地方都可以改用等价的简写名 for——不必担心，Perl 能明白你的意思：

```
for (@lines) { # 打印第一个以 From:开头的行
  print, last if /^From:/;
}
```

2. 用 map 函数从现有列表延展出新列表

如果是从现有列表推导出新列表，请使用 map 函数。后面两行是从文件名列表推演到文件大小列表的代码。请记住，大多数文件测试操作符的默认参数都是 $_（见条款 51）：

```
my @sizes = map { -s } @files;

my @sizes = map -s, @files;
```

map 接受列表上下文参数，第一个参数是用于数据转换操作的表达式或代码块，第二个参数是要遍历的数组或列表。我们常常用它返回空列表或包含多个元素的列表。而在做数据转换时，匹配操作可以写得非常简约直观。

比如接下来的两个例子，map 在做数据转换时，返回模式匹配操作符 m// 内部圆括号捕获的数据。因为这是一个列表上下文，所以 m// 返回的要么是没有匹配时的空列表，要么是匹配时找到的结果集合而成的列表。

下面这个 map 返回以.txt 结尾的文件名的主名，不包括后缀。由于匹配操作位于列表上下文，所以按列表返回圆括号中匹配的内容：

```
my @stem = map { /(.*)\.txt$/ } @files;
```

下面的 map 操作基本类似，从邮件头中找出以 From: 开头的行，并取出其中的邮件地址。只有当成功匹配时，它才会返回相应结果：

```
my ($from) = map /^From:\s+(.*)$/, @message_lines;
```

出于性能考虑，一般我们会直接对默认的 $_ 变量操作。它其实相当于当前列表元素的别名，所以一旦在 map 表达式中修改了 $_ 的内容的话，原始数据也会随之改变。一般认为这种行为不太好，——可谁知道——连你自己都有可能会被这种用法弄昏了头。所以如果你怕搞不清楚，那就记住一个原则，确实要修改原始列表内容的话，就改用 foreach 循环。（稍后就会谈到。）

另外在使用 map 时，得明确它的意义在于返回相应的列表数据，而不是用作控制结构做一堆杂事。下面的例子违反了正确使用 map 的三条原则：第一，tr/// 修改了 $_ 的值，从而改变了

www.TopSage.com

原始数据@elems；第二，map 的返回值没有意义，tr/// 返回的是它在每个元素中删除的数字个数；第三，最后得到的输出结果无用武之地，这和map的设计初衷背道而驰：

```
map { tr/0-9//d } @elems; # 可能有问题
```

虽然下面的例子返回了一个有意义的列表，但 tr///还是在修改@elems（见条款114）：

```
my @digitless = map { tr/0-9//d; $_ } @elems; # 坏习惯
```

如果确实需要在map中使用tr///、s/// 之类的操作，可以复制$_到词域变量再做改动处理。像下面这段代码，就是将$_的值复制到$x后以便再行替换：

```
my @digitless = map {
  ( my $x = $_ ) =~ tr/0-9//d;
  $x
} @elems;
```

3. 用 foreach 修订列表元素内容

如果目的是要修订列表元素内容，请使用 foreach 循环。和map、grep 一样，循环体中的控制变量其实是列表当前元素的别名。所以修改控制变量，实际上就是修改原始数据：

```
# 将列表@nums 中所有元素的值更新为原来的两倍
foreach my $num (@nums) {
  $num *= 2;
}

# 移除数组@ary 各个元素中的数字
foreach (@ary) { tr/0-9//d }

# 使用 s///的版本，更慢一些
foreach (@elems) { s/\d//g }

# 依次将$str1、$str2 和$str3 改成大写
foreach ( $str1, $str2, $str3 ) {
  $_ = uc $_;
}
```

4. 用 grep 筛选列表元素

grep 比较特别，一般用于筛选列表元素或对元素计数。但有时会被滥用，那些觉得 foreach 不如 grep 酷的程序员，喜欢什么循环都用 grep 实现，虽说没什么不可以，但作为头脑清醒的开发人员，以最简单直接的方式使用 grep 才是正道。

下面是 grep 在列表上下文中的常规用法：

```
print grep /joseph/i, @lines;
```

顺便提一下，grep 块的第一个参数，不管是表达式还是代码块，都是在标量上下文中的，这和map不同。这个一般不会对结果有什么影响，但知道总比不知道要好。

在标量上下文中，grep返回符合条件的元素个数，而不是元素列表：

```
my $has_false = grep !$_, @array;
my $has_undef = grep !defined($_), @array;
```

www.TopSage.com

5. 要点

- 在需要从一个列表推演到另一个列表时，应使用 map 函数。
- 在筛选列表元素时，应使用 grep 函数。
- 如果在遍历时需要修改变量，应使用 foreach 函数。

条款21 了解各种字符串引用方法

Perl 有着多种多样的字符串引用方法。

最简单的是单引号方式，除了用于转义的反斜杠和表示字串引用范围的单引号以外，其他字符完全按字面上原本的意义解释：

```
'Isn\'t she "lovely"?' # Isn't she "lovely"?
```

而相比之下，双引号则支持多种表示特殊意义的转义序列。比如常见的有 \t、\n、\r 等，这些都源自 C 语言：

```
"Testing\none\n\two\n\three" # Testing
                             # one     wo
                             #       hree
```

此外还有八进制和十六进制的 ASCII 值转义，比如 \101 和 \x41：

```
"\x50\x65\x72\x6c\x21" # Perl!
```

还可以通过 Unicode 代码点的方式以 \x{} 的写法定义 Unicode 字符（见条款 74）：

```
"There be pirates! \x{2620}";
```

用 charnames 模块的话，还可以按 Unicode 的字符名称表示 Unicode 字符：

```
use charnames;

my $str = "There be pirates! \N{SKULL AND CROSSBONES}";
```

双引号内出现的变量，包括 $ 和 @ 打头的，都会用变量的值替换，我们称为变量内插：

```
foreach $key ( sort keys %hash ) {
  print "$key: $hash{$key}\n";
}
```

数组或列表在做变量内插时，会将所有元素串联起来，以特殊变量 $" 的值作为元素间的分隔符——通常默认为空格字符：

```
my @n = 1 .. 3;
print "testing @n\n"; # testing 1 2 3
```

如果变量名后紧跟其他字符，为有所区别，我们得在变量名标识符前后添上 {}：

```
print "testing @{n}sies\n"; # testing 1 2 3sies
```

而像 \u、\U、\l、\L、\E 这类特殊转义操作符，会在双引号内改变后续字串的大小写：

```
my $v = "very";
print "I am \u$v \U$v\E tired!\n"; # I am Very VERY tired!
```

www.TopSage.com

大家网 TopSage.com

以上介绍的并非全部，变量内插还有许多其他用法，具体细节请查阅 **perlop** 文档中的相关内容。

1. 其他可选引用方式：q、qq 以及 qw

当单引号或双引号频繁出现在字串中时，若能使用其他字符圈引字串，就能免除逐个转义的麻烦，并且看起来也更干净。从原则上来讲，Perl 允许使用任意标点符号作为字串两边的圈引边界，只要不产生歧义就没问题。随便拿一个你喜欢的，放在 q 或 qq 的后面，然后在字串末尾用同样的标点标注结束就可以了。唯一的差别是，qq 相当于双引号，允许变量内插，而 q 相当于单引号，没有变量内插：

```
q*A 'starring' role*  # A 'starring' role
qq|Don't "quote" me!| # Don't "quote" me!
```

如果字串碰巧用到了我们自定义的边界符号，那就必须转义了，否则就无从分辨。不过一般我们可以改用其他符号：

```
q*Make this \*bold\** # Make this *bold*
```

如果用的是可以配成对的字符（无非就是 (、[、< 或 { 之一），那在结束时必须改用对应的结束字符。如下所示，这四种写法完全等效：

```
qq<Don't "quote" me!>
qq[Don't "quote" me!]
qq{Don't "quote" me!}
qq(Don't "quote" me!)
```

Perl 能准确识别嵌套的配对分界符：

```
qq<Don't << quote >> me!> # Don't << quote >> me!
```

2. 用 q{} 或 qq{} 圈引源代码

边界字符的选择，多少会起到体现数据意义的作用，所以对源代码，最好能用花括号来圈引，既符合习惯，又干净明了。比如用 Benchmark 测试代码性能时，目标代码的引用：

```
use Benchmark;

our $b = 1.234;
timethese(
  1_000_000,
  {
    control => q{ my $a =     $b },
    sin     => q{ my $a = sin $b },
    log     => q{ my $a = log $b },
  }
);
```

3. 用 qw() 构造列表可免除逗号和引号的使用

对于快速创建字符串列表，我们可以用 qw（quote words）这样的形式一次取得所有单词字串列表。Perl 会用空格符分割 qw 内字串：

```
@ISA = qw( Foo Bar Bletch );
```

前面的写法，比起后面这种传统写法来，要简捷明快得多，因为用于区分单词的逗号和引号也都省掉了：

```
@ISA = ('Foo', 'Bar', 'Bletch');
```

不要在 qw 操作符内误用逗号，它会被当成是字串的一部分。以下面代码为例，最终得到的是“Foo,”、“Bar,”和“Bletch”这三个字串列表。除了最后一个“Bletch”没改变外，其余字串末尾都多了一个逗号：

```
@ISA = qw( Foo, Bar, Bletch );
```

如果启用 warnings 的话（见条款 99），就会看到这样的警告：

```
Possible attempt to separate words with commas ...
```

4. 另一种引用方式：here doc 字符串

Perl 的 **here doc**（或全称 **here document**）字串提供了圈引大段文本的另一种途径。可能许多人已经对此非常熟悉了——Perl 的这个功能源自于 UNIX 中 shell 的 here doc 特性。一般对于大段文本或源代码，我们都会选用 here doc 的方式圈引。

here doc 字符串从 << 加一个标识符名称之后的一行开始，一直到出现该标识符开头的独立行之前结束。如果标识符两边使用引号（包括单引号、双引号或反引号），那么引号的类型决定了 here doc 引起的字符串是否允许变量内插等。如果标识符两边没加引号，则默认使用双引号：

```
print <<EOT; # ;表示该语句结束，所以这里的注释不包含于该字符串中
Dear $j $h,

You may have just won $m!
EOT
```

如果不希望变量内插，可以在标识符两边使用单引号：

```
print <<'XYZZY'; # 这次用单引号……
Dear $j $h,

You may have just won $m!
XYZZY
```

由于一次允许使用多个 here doc，所以下面这样的写法看起来有些奇怪。实际上它分别定义了两段字串：

```
print <<"HERE", <<"THERE";
This is in the HERE here doc
HERE
This is in the THERE here doc
THERE
```

要是把 here doc 用作子程序参数，看起来也会很奇怪，但本质上都是一样的，只不过把大段文本提取到子程序调用语句之外而已：

```
some_sub( <<"HERE", <<"THERE" );
This is in the HERE here doc
HERE
```

www.TopSage.com

```
This is in the THERE here doc
THERE
```

你当然可以这么写，但考虑到让其他人更容易理解，或许避免这种写法更妥当些。

5. 要点

- 用 q() 或 qq() 圈引普通字串。
- 用 qw() 自动圈引单词列表。
- 用 here doc 圈引多行文本。

条款22　掌握多种排序方式

在Perl最基本的排序操作中，所用的代码相当简洁。仅仅用一个 sort 操作符，就能对列表进行排序。它会返回排序后得到的新列表：

```
my @elements = sort qw(
  hydrogen
  helium
  lithium
);
```

Perl默认按UTF-8排序规则进行排序（除非指定用 use locale 或 use bytes），也就是说，排序时依照UTF-8设定的各字节的码值（编码后的字节数值），从第一个元素的第一个字节起比较，然后第二个，第三个，再到后一个元素的各个字节，只要还未区分出大小前后，就会一直逐个比较下去。不过这种规则要求我们对于字符的定义，得先有个准确理解（见条款77）。

因此在对数字或大小写混杂的字串排序时，会看到意外结果，可以用后面的例子试一试：

```
print join ' ', sort 1 .. 10;
print join ' ', sort qw( my Dog has Fleas );
```

如果不想用默认的UTF-8排序，那你就需要自己编写用于比较的子程序。

1. 比较（sort）子程序

所谓的Perl排序子程序，实际上并非完整的排序算法，比较恰当的说法是“比较子程序”。

和一般的子程序不同，排序子程序得到的参数是经过硬编码的两个特殊包变量 $a 和 $b，而非 @_ 。$a 和 $b 在排序子程序内部是本地化的，这就好比是在子程序内部开头的地方，总有一句无形的 local($a, $b) 语句一样（但不用麻烦我们自己写）。

$a 和 $b 能得到 use strict vars 的特别照顾——使用之前无需特别声明（见条款3）。它们属于当前包，而不像特殊变量那样只属于 main 包。

Perl在排序过程中会不断调用这个 sort 子程序。它的任务是比较 $a 和 $b 的大小，然后根据 $a 小于 $b、$a 等于 $b、$a 大于 $b 这三种情况分别返回 -1、0、1。

Perl内建的排序方式相当于用 cmp 操作符比较。下面是通过具名子程序的方式，在 utf8ly 内指定排序规则：

```
sub utf8ly { $a cmp $b }
```

www.TopSage.com

```
my @list = sort utf8ly @list;
```

而更恰当的方式，是在 sort 后面使用代码块。直接将原来子程序中的内容移过来就好了：

```
my @list = sort { $a cmp $b } ( 16, 1, 8, 2, 4, 32 );
```

$a 和$b 的比较方法决定了最终得到的排序顺序。比如把 cmp 操作符换成 <=>，就变成了按数字大小排序：

```
my @list = sort { $a <=> $b } ( 16, 1, 8, 2, 4, 32 );
```

还可以进行大小写无关的排序，只要先把字符串全部转换成小写再作比较就可以了：

```
my @list = sort { lc($a) cmp lc($b) } qw(This is a test);

# ('a', 'is', 'test', 'This')
```

而对换 $a 和 $b 的位置就能得到倒序结果：

```
my @list = sort { $b cmp $a } @list;
```

当然也可以根据$a 和$b 的值计算后再作比较，比如接下来的这条代码是按文件最后修改时间排序：

```
my @list = sort { -M $a <=> -M $b } @files;
```

我们时常需要根据散列值的大小对散列键进行排序。可以借助 $a 和 $b 取得散列值做比较：

```
my %elems = ( B => 5, Be => 4, H => 1, He => 2, Li => 3 );

sort { $elems{$a} <=> $elems{$b} } keys %elems;
```

甚至还可以按照多个键先后排序。还记得逻辑短路操作符 or 的妙用吗？正如接下来这段代码所展示的，若是姓相同，则再按名字排序：

```
my @first = qw(John Jane Bill Sue Carol);
my @last = qw(Smith Smith Jones Jones Smith);

my @index = sort {
  $last[$a] cmp $last[$b]  # 先按姓排序，然后
    or
  $first[$a] cmp $first[$b]     # 按名排序
} 0 .. $#first;

for (@index) {
  print "$last[$_], $first[$_]\n"; # Jones, Bill
}  # Jones, Sue
   # Smith, Carol, etc.
```

前例中，实际是根据数组下标取得的数据进行排序，这是很常见的手法。请注意其中 or 的使用，每次 Perl 调用比较子程序时，会先计算 or 左边的表达式，如果结果是非零真值（按此例来讲，就是当 $a 和 $b 不相同时），or 直接结束，返回左边的求值结果。否则，它会继续计算右边的表达式，并返回它的结果。

请注意，若是 Perl 的 or 操作符（或优先级更高的||操作符）只能返回数字 1 或 0 两种结果

www.TopSage.com

的话就不可能这么用了。由于 or 返回的，实际上是左边表达式或右边表达式的计算结果，这种用法才成立。

2. 高级排序：一般做法

有时在比较两个值的时候需要进行大量计算，比如按照密码文件的第三个字段（也就是uid字段）来排序：

```
open my ($passwd), '<', '/etc/passwd' or die;
my @by_uid =
  sort { ( split /:/, $a )[2] <=> ( split /:/, $b )[2] }
  <$passwd>;
```

乍看之下，这段程序没什么问题，它确实能按指定字段排序。但每次比较都要执行两次复杂的 split 计算，而拆分方式又毫无分别，显然可以进一步优化性能。

对于每个要比较的元素，sort 都会运行一次或多次子程序。假设需要排序的元素个数为 n，那么排序时子程序的调用次数就是 n*log(n)。所以为了提升排序的整体效率，就要想办法提高每次比较的速度。如果比较之前需要进行复杂计算，应该考虑使用缓存机制，在比较前就先计算好要比较的数据，比较时直接使用缓存，以提高排序速度。

针对前例，在作具体比较前先创建一个散列表，以原始数据行作为关键字，而以拆分出来的用户标识号 uid 作为值，然后对散列排序：

```
open my ($passwd), '<', '/etc/passwd' or die;
my @passwd = <$passwd>;

# 整个一行成为键，而值是 uid
my %lines = map { $_, ( split /:/ )[2] } @ passwd;

my @lines_sorted_by_uid =
  sort { $lines{$a} <=> $lines{$b} } keys %lines;
```

3. 高级排序：更酷的做法

不管是经过深思熟虑，还因为是随意尝试或灵光乍现，久而久之 Perl 程序员在持续的实践中逐渐积累起一套实现复杂排序变形的惯用习语。

上面那个实现方式的主要缺点在于，它需要额外准备一条语句，建立辅助排序的数组或散列。我们有一个回避这种用法的技巧，称为 **Orcish Maneuver**（或者 || cache），它充分利用了不为人熟知的||=操作符的特性。让我们从之前按文件最后修改日期排序的例子开始，看看如何改进。

● 用 Orcish Maneuver *技巧排序*

下面是之前实现的方式，由于多次重复计算 -M，所以性能不佳：

```
my @sorted = sort { -M $a <=> -M $b } @files;
```

而改用 Orcish Maneuver 技巧后的写法如下：

```
my @sorted =
  sort { ( $m{$b} ||= -M $b ) <=> ( $m{$b} ||= -M $b ) }
  @files;
```

这段代码究竟表示什么？请别急，先了解一下 ||= ：

www.TopSage.com

TopSage.com


```
$m{$a} ||= -M $a
```

这是一种简写，它实际上等同于下面这条语句：

```
$m{$a} = $m{$a} || -M $a
```

在排序子程序第一次遇到某个文件 $a 时，对应的 $m{$a} 的值是 undef，也就是假，于是 ||= 右边的表达式 -M $a 求值后存入 $m{$a}。请注意，Perl 中的 || 和 C 语言的不同，它返回的是右边表达式的计算结果，而非简单的数字 0 或 1（真或假）。所以，此刻计算结果得到缓存。

之后再对该文件取最后修改时间作比较时，就会直接使用第一次比较时保存到 $m{$a} 中的值。

这个 %m 散列是临时变量，在排序开始前应该是空的，或处于未定义状态。为确保没有干扰，我们可以把该变量的声明放在排序之前，并用花括号限定作用域，类似这样：

```
{
  my %m;
  @sorted = sort ...
}
```

● *施瓦茨变换*

目前为止最精简的全能排序技术还得算是**施瓦茨变换**（Schwartzian Transform），名字来自发明人 Randal Schwartz。简而言之，施瓦茨变换就是接续若干 map 实现的 sort 排序。

为说明施瓦茨变换，让我们一步步来看数据的辗转变化过程。还是借之前按文件最后修改时间排序的例子来说，但这次改造后的性能会更高。第一件要做的事情，是取得文件名列表：

```
my @names = glob('*');
```

然后，将文件名列表转化为相同长度的数组列表：

```
my @names_and_ages = map { [ $_, -M ] } @names;
```

列表中每个元素都是一个匿名数组，每个数组包含两个子元素——可以称这样的数组为二元组（见条款 13）。第一个子元素是原始文件名（来自于 $_），第二个子元素是最后修改时间，单位为天（针对隐含的默认参数 $_，通过 -M 计算的结果）。

下一步，在 sort 块内根据列表中匿名数组的第二个元素按数字大小排序：

```
my @sorted_names_and_ages =
  sort { $a->[1] <=> $b->[1] } @names_and_ages;
```

在 sort 内的 $a 和 $b，实际代表的是 @names_and_ages 的元素，也就是数组引用。所以 $a->[1] 表示二元组的第二个元素，即文件最后修改日期到今天的天数。而最终的结果是对二元组的天数按升序进行数字排序（因为这里用的是 <=> 操作符）。

基本步骤就是这些，最后要做的无非是从二元组中提取文件名。这个步骤非常简单，一次 map 即可：

```
my @sorted_names =
   map { $_->[0] } @sorted_names_and_ages;
```

好了，所需步骤就这么多。但上面逐行递推的方式太过繁琐，可以组装起来并为一条语句，成为真正的施瓦茨变换：

```
my @sorted_names =
```

www.TopSage.com

```
map { $_->[0] }     # 4. 提取原始文件名
sort { $a->[1]
       <=>
       $b->[1] }    # 3. 对[name, key]二元组排序
map { [ $_, -M ] } # 2. 创建[name, key]二元组
@files;             # 1. 原始数据
```

从下往上阅读，就能看出这段代码对应于之前分开写的每条步骤——只不过现在都堆叠起来省掉了临时变量。

单个元素的排序也可以借鉴这个思路，只需要在匿名数组中修改右边的元素值，并调整所需要的比较操作符。下面是密码文件按第三个字段提取uid后按大小排序的代码，也是用施瓦兹变换实现的：

```
open my ($passwd), '<', '/etc/passwd' or die;
my @by_uid =
  map  { $_->[0] }
  sort { $a->[1] <=> $b->[1] }
  map  { [ $_, ( split /:/ )[2] ] } <$passwd>;
```

显而易见，这比先前的要简明得多。借助施瓦兹变换直接从文件句柄读取数据，不但可以省去对缓存数据的临时变量%key的使用，还可以省去@passwd数组。

4. 要点

- Perl默认按UTF-8规则排序。
- 可以让sort使用自定义的排序子程序。
- 为提高运算复杂的排序程序的性能，可以采取先缓存键值后排序的方式。

条款23　通过智能匹配简化工作

Perl 5.10引入了智能匹配操作符~~。它能根据两端算子的类型决定实际的匹配方式。使用智能匹配可以用最少的指令写出功能强大的条件判断式。完整的使用细则可见**perlsyn**文档，这里会列出一些有趣的案例。智能匹配操作符通常和given-when一起联合使用（见条款24）。

启用智能匹配，需确保Perl版本在5.10.1以上。智能操作符在设计之初是符合交换律的，也就是说两边的算子可以对调，但后来做了修订，交换左右算子会改变匹配行为。所以为了避免误解，本条款中所有例子都预先启用Perl 5.10.1版本：

```
use 5.010001;
```

先来看一看有关检查散列键或数组元素是否存在的例子。任务很简单，写起来也不复杂：

```
if ( exists $hash{$key} ) { ... }

if ( grep { $_ eq $name } @cats ) { ... }
```

用智能匹配改写，代码看起来更漂亮，而且尽管任务不同，形式上也能保持一致：

```
use 5.010001; # ~~的行为自 Perl 5.10.1 起被修订了

if ( $key ~~ %hash ) { ... }
```

www.TopSage.com

```
if ( $name ~~ @cats ) { ... }
```

接下来，考虑用正则表达式检查散列键，但好像只能手工逐个遍历：

```
my $matched = 0;

foreach my $key ( keys %hash ) {
  do { $matched = 1; last } if $key =~ /$regex/;
}
if ($matched) {
  print "One of the keys matched!\n";
}
```

这样写很啰嗦。其实可以把这些逻辑封装到某个子程序内部实现，实际上，在 Perl 5.10.1 中，这些工作早已内建完毕，直接拿来测试即可：

```
use 5.010001;

if ( %hash ~~ /$regex/ ) {
  say "One of the keys matched!";
}
```

就算检查的是数组，写法也大体相同：

```
use 5.010001;

if ( @array ~~ /$regex/ ) {
  say "One of the elements matched!";
}
```

其他通过智能匹配可以大大简化代码的操作有：

```
%hash1 ~~ %hash2       # 键相同的两个散列

@array1 ~~ @array2     # 完全相同的两个数组

%hash ~~ @keys         # 数组中是否存在%hash 中的某个键

$scalar ~~ $code_ref   # $code_ref->( $scalar ) 是否返回真
```

有关智能匹配操作符的完整功能列表，请参考 **perlsyn** 文档。

要点

- 用智能匹配封装常见的复杂匹配模式。
- 根据算子类型，智能匹配操作符会自动匹配相应的行为。
- 请限定 Perl 5.10.1 以上版本，以确保稳定的智能匹配行为。

条款 24　用 given-when 构造 switch 语句

大概从 Perl 初生的那天起，Perl 程序员就没停止过抱怨：其他都有了，为什么不把 C 语言的 switch 语句也借来用用？这个愿望一直到 Perl 5.10 才得以实现，Perl 不仅引入了相同语法，还把它改造得和 Perl 里面其他东西一样——酷酷的非比寻常。

www.TopSage.com

1. 更少的输入

Perl给switch起了一个新名字，叫作given-when。这个名字符合人们谈到这一操作时的普遍说法："我给出（given）某个条件，当（when）实际情况吻合，操作如下。"你也许已经会用一组if-elsif-else语句表达：

```
my $dog = 'Spot';

if    ( $dog eq 'Fido' )  { ... }
elsif ( $dog eq 'Rover' ) { ... }
elsif ( $dog eq 'Spot' )  { ... }
else                      { ... }
```

根据$dog种类，需要运行不同代码。这种老式写法要求每个条件分支都重复一遍相近的判断语句。对于Perl这样的高级语言来说，这么写实在有点大材小用。事实上，可以使用given-when来代替这种写法，而改写后的代码如下：

```
use 5.010;

given ($dog) {
  when ('Fido')  { ... }
  when ('Rover') { ... }
  when ('Spot')  { ... }
  default        { ... };
};
```

这能帮你省掉不少敲打键盘的功夫！因为Perl在幕后悄悄完成了几项任务：首先，将$dog**设置为当前话题**，也就是把$dog这个变量赋给$_，这样就不必反复输入$dog；然后，Perl会自动完成$_和给定数据间的比较。下面是实际展开后的代码，如果你不怕麻烦的话，也可以直接这样写：

```
use 5.010;

given ($dog) {
  my $_ = $dog;
  when ( $_ eq 'Fido' )  { ... }
  when ( $_ eq 'Rover' ) { ... }
  when ( $_ eq 'Spot' )  { ... }
  default { ... };
};
```

2. 智能匹配

given-when结构不只是简单的串联条件判断式。之前的例子用的是显式的字符串eq比较。而在更靠前的例子中，Perl又是如何知道该用字符比较操作？实际上，除非你明确指定比较操作符，默认情况下，Perl总是自动使用智能匹配操作符~~（见条款23）：

```
use 5.010;

given ($dog) {
  when ( $_ ~~ 'Fido' )  { ... }
```

www.TopSage.com

```
  when ( $_ ~~ 'Rover' ) { ... }
  when ( $_ ~~ 'Spot' )  { ... }
  default { ... };
};
```

智能匹配能根据算子数据类型自动选择合适的比较方法。在这段代码中 Perl 看到的是标量 $_ 和字符串 'Fido'，所以决定使用字符串比较。任何地方都可以使用智能匹配，而如果在 when 语句中不加指定，默认用的就是智能匹配。

如果智能匹配时你给定的数据类型不同，得到会是与数据类型相对应的智能匹配行为。**perlsyn** 文档列出了所有类型的比较，以下列选部分比较有趣的：

```
$dog  ~~ /$regex/    # $dog 能被正则匹配

$dog  ~~ %Dogs       # $dog 是 %Dogs 中的键

$dog  ~~ @Dogs       # $dog 是 @Dogs 中的元素

@Dogs ~~ /$regex/    # @Dogs 中至少有一个元素能与正则匹配

%Dogs ~~ /$regex/    # %Dogs 中至少有一个键能与正则匹配
```

在 when 中简单给出测试条件的写法，实际是将被比较的目标（即当前主题 $_）放在智能匹配的左侧，把测试条件放在右侧，完成智能匹配。所以，下面两条语句实际是等效的：

```
when (RHS) { ... }

when ( $_ ~~ RHS ) { ... }
```

3. 多分支处理

默认情况下，只要某个 when 块得到匹配，这段程序的运算就算结束了。Perl 不会再计算其他 when 块，这点和 if-elsif-else 结构相似。这就好比是在每个 when 块末尾都有一个隐形的 break 语句，用以直接跳出整个循环：

```
use 5.010;

given ($dog) {
  when ('Fido')  { ...; break }
  when ('Rover') { ...; break }
  when ('Spot')  { ...; break }
  default        { ... };
};
```

当然，利用 continue 语句，就可以使程序在当前 when 块运算结束后进入下一个 when 块继续比较。在后面这个例子中，能完成针对 $dog 名称而进行的所有 when 测试：

```
use 5.010;

my $dog = 'Spot';

given ($dog) {
  when (/o/) { say 'The name has an "o"'; continue }
  when (/t/) { say 'The name has a "t"'; continue }
```

www.TopSage.com

大家网 TopSage.com

```
  when (/d/) { say 'The name has a "d"'; continue }
};
```

4. 代码组合

if-elsif-else结构还有一个缺陷是，它不能在两个条件中组合其他代码。任何代码，都必须在找到匹配条件后才能执行。而在given-when结构中，你可以在when块之间自由插入任意代码，哪怕是中途修改主题变量的也没问题：

```
use 5.010;

my $dog = 'Spot';

given ($dog) {
  say "I'm working with [$_]";
  when (/o/) { say 'The name has an "o"'; continue }

  say "Continuing to look for a t";
  when (/t/) { say 'The name has an "t"'; continue }

  $_ =~ tr/p/d/;
  when (/d/) { say 'The name has an "d"'; continue }
};
```

5. 对列表进行分支判断

在foreach循环中我们也能用when，这和在given中相似，只不过它是依次从列表取测试目标：

```
use 5.010;

my $count = 0;

foreach (@array) {
  when (/[aeiou]$/)  { $vowels_count++ }
  when (/[^aeiou]$/) { $count++ }
}

say "\@array contains $count words ending in consonants and
  $vowel_count words ending in vowels";
```

6. 要点

- 如果需要写分支语句，请使用given-when结构。
- when语句中的智能匹配操作，默认用的是$_变量。
- 除given外，其他循环结构中也能使用when语句。

条款25　用do {}创建内联子程序

do {}这种语法能把几条语句组合成单条表达式，这有点类似于内联子程序。比如下面这段代码，在读取某个文件所有内容之前，先将表示输入数据分隔符的$/变量本地化（见条款43），

然后通过词法文件句柄打开文件（见条款 52），读取数据返回后，存入标量变量。所有这一切，都在同一个语句块中完成：

```
my $file = do {
  local $/;
  open my ($fh), '<', $filename or die;
  <$fh>;
};
```

因为有了 do 语句限定代码作用域，其中 local 和 my 声明的变量也都随之被限定在内，和外部代码完全不相干。最后一条表达式 <$fh> 会作为代码块的返回值返回，这和传统的子程序类似，最后将返回值保存到外部变量 $file。

下面看看若是不用 do {} 写出来的代码会是什么样子。由于代码块中的数据生命期有限，所以为了让计算结果得以传递到外部，我们不得不在最后以完整的赋值语句作为结束。这样加上开始时的变量声明，就得敲上两遍 $file，既麻烦又累赘：

```
my $file;
{
  local $/;
  open my ($fh), '<', $filename or die;
  $file = join '', <$fh>;
}
```

这种借助 if-elsif-else 结构返回不同数据的写法，也可以归纳为 do {} 的形式，不但能省略了大量重复代码，语义也更为清晰。举个例子，根据地区语言的不同，你也许需要选择适合的千位分隔符和小数点表示方式。一般总是先声明这两个分隔符变量，然后根据不同条件分别赋值：

```
my ( $thousands_sep, $decimal_sep );
if ( $locale eq 'European' ) {
  ( $thousands_sep, $decimal_sep ) = qw( . , );
}
elsif ( $locale eq 'English' ) {
  ( $thousands_sep, $decimal_sep ) = qw( , . );
}
```

这样的代码显得非常笨拙。重复输入的变量，相类似的赋值结构，无形之中模糊了原本的意图，厚重且拖沓。而借助 do 语句，利用最后求值的表达式即返回值这一特性，合并目标变量，代码的意义就会变得直接明确：

```
my ( $thousands_sep, $decimal_sep ) = do {
  if     ( $locale eq 'European' ) { qw( . , ) }
  elsif  ( $locale eq 'English' ) { qw( , . ) }
};
```

有时候我们也会借助 do 来弱化错误处理的相关代码。比如打开一组文件，但不希望在某次打开失败时就立即中止，我们可以在 or 右端接续一个 do 语句，在其内部处理发出警告的任务，并随后指示它继续运行：

```
foreach my $file (@files) {
  open my ($fh), '<', $file or do { warn ...; next };
  ... do stuff ...;
}
```

www.TopSage.com

要是 do {} 块里的内容太长，最好还是把它写成一个单独的子程序。原本就是为了方便而使用的，但一味滥用的话就违背这个习语的初衷了。

do 还有一个用途，但比较少见，该特定情形就是在第一次 while 条件判断之前，需要先运行一次循环内容才能得到数据以作后续条件判断。比如不断提示用户输入，直到特定信息到来时再终止的程序。这种情况下，必须先取得第一次输入，然后才能作条件测试，决定继续提示还是终止，显然，这么写又出现了代码重复：

```
print "Type 'hello': ";
chomp( my $typed = <STDIN> );
while ( $typed ne 'hello' ) {
  print "Type 'hello': ";
  chomp( $typed = <STDIN> );
}
```

通过使用条件后置的 while 循环，先执行 do {}，再判断是否需要继续提示用户输入：

```
my $typed;
do {
  print "Type 'hello': ";
  chomp( $typed = <STDIN> );
} while ( $typed ne 'hello' );
```

这里的$typed 变量必须声明，虽然还是不够美观，但至少比起之前的写法还是要简洁得多。

要点

- do 语句块返回最后一个表达式求值的结果。
- 任何能使用表达式的地方都可以使用 do 语句块。
- 可以利用 do 语句块来限制变量的作用域。

条款 26　用 List::Util 和 List::MoreUtils 简化列表处理

掌握列表的各项操作，是深入理解 Perl 的必经之路。事实上，在 *Higher Order Perl* 这本书的序言中，Mark Jason Dominus 就曾说过，Perl 其实更接近 Lisp 语言，而不是 C。此话不假，Perl 有内置的 map、grep 和 foreach 语句，组合起来能完成许多复杂的列表处理。而有些列表操作极其常见，与其重复发明轮子，还不如直接用 List::Util 或 List::MoreUtils 这两个 C 语言实现的模块提供的函数来得更快。

1. 快速查找最大值

查找列表中的最大值，你自己写的话也不算太麻烦：

```
my @numbers = 0 .. 1000;
my $max     = $numbers[0];
foreach (@numbers) {
  $max = $_ if $_ > $max;
}
```

不过纯粹用 Perl 实现，相对来说性能要差些。而 Perl 自带的 List::Util 模块提供了 C 语言

www.TopSage.com

实现的 max 子程序，可以直接返回列表中的最大值：

```
use List::Util qw(max);

my $max_number = max( 0 .. 1000 );
```

此外还有一个 maxstr 子程序，能返回列表中最大的字符串：

```
use List::Util qw(maxstr);

my $max_string = maxstr( qw( Fido Spot Rover ) );
```

类似的，对列表中所有数字求和，若由你自己写，就可能写成这样：

```
my $sum = 0;
foreach ( 1 .. 1000 ) {
  $sum += $_;
}
```

而用 List::Util 提供的 sum 子程序则要简单快捷得多：

```
use List::Util qw(sum);

my $sum = sum( 1 .. 1000 );
```

2. 列表归并

对一系列数字求和的办法还有一个，就是利用 List::Util 提供的 reduce 函数逐项迭代。它也是用 C 语言实现的，所以执行起来很快，语法也更明晰：

```
use List::Util qw(reduce);

my $sum = reduce { $a + $b } 1 .. 1000;
```

与 sort 类似，reduce 也以代码块作为参数，不过运行机制稍有不同。每次迭代，它会先从参数列表中取出前两个元素，分别设置别名 $a 和 $b，这样参数列表的长度就会缩短两个元素。然后 reduce 把语句块返回的计算结果再压回到参数列表的头部。如此往复，直到最后列表里只剩下一个元素，也就是迭代的计算结果 $sum。

了解工作原理后，就能灵活实现某些尚未定义成标准函数的计算方式，比如对列表元素逐项累乘：

```
my $product = reduce { $a * $b } 1 .. 1000;
```

3. 判断是否有元素匹配

纯粹用 Perl 实现的话，找出列表中第一个符合某项条件的元素，比找出所有符合条件的要麻烦一些。比如下面这条语句用于判断是否有大于 1000 的元素，它写起来倒是很容易：

```
my $found_a_match = grep { $_ > 1000 } @list;
```

但若是 @list 有一亿个元素，而第一个元素就是 1001 呢？上面的代码照旧会逐项检查每一个元素，而答案其实早就知道了。当然，我们可以自行控制退出循环，但有谁会愿意每次都写上这么一大段代码？

www.TopSage.com

```
my $found_a_match = 0;
foreach my $elem (@list) {
  $found_a_match = $elem if $elem > 1000;
  last if $found_a_match;
}
```

List::Util的first子程序就是为了解决这样的问题而设计的，并且它会告诉你第一个找到的元素是什么。它不会浪费时间做无用功，一旦找到答案，就会立即停止扫描，而不会为了找出那一个大于1000的元素去扫描整个列表：

```
use List::Util qw(first);

my $found_a_match = first { $_ > 1000 } @list;
```

此外，在CPAN的List::MoreUtils模块中，也提供了许多额外的实用函数下载：

```
use List::MoreUtils qw(any all none notall);

my $found_a_match = any    { $_ > 1000 } @list;
my $all_greater   = all    { $_ > 1000 } @list;
my $none_greater  = none   { $_ > 1000 } @list;
my $all_greater   = notall { $_ % 2    } @list;
```

4. 一次遍历多个列表

有时候我们手上会有几个相互关联的列表需要同时遍历。最普通的做法是利用数组下标，同步提取对应元素，计算后存入另一个列表。比如下面这段代码，它依次取出数组@a和@b中的元素，求和后存入数组@c相同的位置。每次迭代都要通过数组下标定位，代码比较复杂：

```
my @a = (...);
my @b = (...);
my @c;

foreach my $i ( 0 .. $#list ) {
  my ( $a, $b ) = ( $a[$i], $b[$i] );
  push @c, $a + $b;
}
```

或者，用List::MoreUtils的pairwise子程序，以更漂亮的方式实现同步叠加：

```
use List::MoreUtils qw(pairwise);

my @c = pairwise { $a + $b } @a, @b;
```

pairwise子程序只适用于两个列表的同步计算，对于三个及以上的列表，可选用相近的each_array子程序。虽然写起来好像有些复杂，不过比起纯粹自己实现，已经相当简单了：

```
use List::MoreUtils qw(each_array);

my $ea = each_array( @a, @b, @c );

my @d;
while ( my ( $a, $b, $c ) = $ea->() ) {
  push @d, $a + $b + $c;
}
```

www.TopSage.com

TopSage.com

5. 数组合并

合并多个数组的操作虽然也可以自己写，但终究不如用 List::MoreUtils 的 mesh 子程序方便：

```
use List::MoreUtils qw(mesh);

my @odds = qw/1 3 5 7 9/;
my @evens = qw/2 4 6 8 10/;

my @numbers = mesh @odds, @even; # 返回 1 2 3 4 ...
```

6. 其他更多函数

以上只是介绍了 List::Util 和 List::MoreUtils 模块提供的几个常用函数，除此之外还有许多函数可以简化列表操作，不光是输入的代码变少了，程序要表达的逻辑也更清晰。至于这些函数的具体用途，还请自行参考相关模块文档。

7. 要点

- 使用 List::Utils 和 List::MoreUtils 模块简化日常列表处理。
- 用 List::MoreUtils 的 all、any、none 或者 notall 函数筛选列表元素。
- 用 pairwise 或 each_array 完成多个列表的同步处理。

条款 27　用 autodie 简化错误处理

Perl 有许多内置函数都是系统调用，可能会因为不可控制的原因而失败，所以有必要对最终运行结果作检查。比如尝试打开某个文件，在使用文件句柄前，得先确认打开文件的操作是成功的：

```
open my ($fh), '<', $file
  or die "Could not open $file: $!";
```

不管怎么说，尽管加上错误处理的代码会让整个程序看起来比较乱，但这些代码是必要的。有一种办法可以避免手工输入这类代码，即用 autodie 编译指令（从 Perl 5.10.1 起开始自带，也可以直接从 CPAN 安装）：

```
use autodie;

open my ($fh), '<', $file;      # 会自动加入 die 相关的检查
```

默认情况下，autodie 会对它能起作用的所有函数生效，这包括绝大多数内建的同系统底层交互的函数。如果只是希望对某些特定函数起作用，可以将各个函数的名字或一组函数的组名列出来告诉 autodie：

```
use autodie qw(open close);     # 只对特定函数生效

use autodie qw(:filesys);       # 只对某组函数生效
```

www.TopSage.com

如果只需要小范围内启用 autodie，也可以把它用在代码块内作为词法编译指令。这和 strict 编译指令用法类似：

```
{
  use autodie;

  open my ($fh), '<', $file;  # 会自动加入 die 相关的检查
}
```

当然，相对的，也可以全局设置为启用 autodie，到个别代码块中再临时关闭：

```
use autodie;
{
  no autodie; # 在这个块中回到普通方式

  chdir('/usr/local') or die "Could not change to $dir";
}
```

在 autodie 捕获到错误时，它会把 $@ 设置为 autodie::exception 对象，而 $@ 就是表示 eval 错误的变量：

```
use autodie;

eval { open my ($fh), '<', $file };
my $error = $@; # 必须立刻提取$@，以防其他错误覆盖
```

接下来我们可以通过查询错误对象的信息，探求问题出现的根源。autodie 非常聪明，它能把错误类型事先进行归类，以便在后续操作中灵活处理。autodie::exception 能感知智能匹配操作符（见条款23）的运算，并以相协调的方式工作，所以我们可以借此写出条理清晰、层次分明的错误处理代码。下面是典型的按照特定顺序检测错误消息或错误类型的代码（具体类型请参考 autodie 文档）：

```
use 5.010;
use autodie;

eval { open my ($fh), '<', $file };
my $error = $@; # 必须立刻提取$@，以防其他错误覆盖

given ($error) {
  when (undef)   { say "No error"; }
  when ('open')  { say "Error from open"; }
  when (':io')   { say "Non-open, IO error."; }
  when (':all')  { say "All other autodie errors." }
  default        { say "Not an autodie error at all." };
}
```

如果手头没有 Perl 5.10 以上的版本，也就是说无法使用智能匹配操作，那么处理 autodie 的错误就难免复杂一些：

```
# 以下代码来自 autodie 文档
if ( $error and $error->isa('autodie::exception') ) {
  if ( $error->matches('open') ) {
    print "Error from open\n";
```

www.TopSage.com

```
  }
  if ( $error->matches(':io') ) {
    print "Non-open, IO error.\n";
  }
}
elsif ($error) { # 不是 autodie 抛出的异常
  ...;
}
```

要点

- 可以用 autodie 自动处理内建函数报出的错误。
- autodie 起作用的代码范围和函数范围是可以限定的。
- 用 eval 语句捕获 autodie 抛出的异常。

www.TopSage.com

第3章

正则表达式 3

Perl的正则表达式本身就相当于一门语言了，而且这门语言甚至比Perl更复杂。

我们不会用到正则表达式的全部特性，但其中有些特性的确能大大简化日常工作。本章介绍其中较为常用的一部分特性。

尽管为了提升效率，开发人员已经对正则表达式引擎内部作了大量优化，但即使经验丰富的开发人员，也偶尔会写出效率极为低下的匹配和替换表达式。

效率并不总是我们的首要目标。事实上，在软件开发过程中，效率不应该是我们的首要目标。通常来说，程序员的首要任务是提供完备、稳定而正确的解决方案。当然在开发时，时刻注意效率问题总还是没有坏处的。

由于现在Perl已经能够处理Unicode字符，所以Perl的正则表达式同样能够处理字节（byte）、字符（character）和字素（grapheme）。不仅如此，正则表达式还可以处理字符属性（character property）。我们把大部分此类问题保留到第8章再详细讨论，也就是说，第8章也将包含不少与正则表达式相关的内容。

条款28 了解正则表达式操作符的优先级

“正则表达式”一词中之所以包含“表达式”，是因为构成和解析正则表达式的语法近似于算术表达式。诚然，正则表达式与算术表达式的作用各不相同，但理解二者的相似性，有助于写出更严谨的正则表达式，或者说更完美的Perl程序。

正则表达式由原子和操作符组成。**原子**（atom）是构成正则表达式的基本单位，通常是指仅匹配单个字符的匹配模式。例如：

```
a       # 匹配字母 a
\$      # 匹配字符 $
\n      # 匹配换行符
[a-z]   # 匹配任何一个小写字母
.       # 匹配除 \n 以外的任意字符
\1      # 反向引用所匹配到的第一组捕获内容（文本长度不限）
```

此外还有一些特殊的“零宽度”原子[①]，例如：

① 实际匹配的不是某个字符，而是字符之间的变化状态。——译者注

www.TopSage.com

```
\b      # 单词边界，从 \w 转换到 \W 的分界点，反之亦然
^       # 匹配字串行首位置
\A      # 字串的绝对行首
\Z      # 字串末尾位置或换行符位置
        # 可能是也可能不是零宽度
\z      # 绝对行尾位置，之后再无其他内容
```

原子由正则表达式操作符修饰或联结在一起。与算术表达式相似，正则表达式的操作符之间也是有优先级次序的。

1. 正则表达式的优先级

幸运的是，正则表达式只有四层优先级。试想一下如果正则表达式的优先级和数学表达式一样多一样复杂，会是怎样一种情况！

圆括号和其他分组操作符拥有最高优先级。表 3-1 列出了正则表达式操作符的优先级。

量词与它所修饰的元素的结合最为紧密，不论所修饰的是原子还是分组：

```
ab*c        # 匹配 ac、abc、abbc、abbbc 等
abc*        # 匹配 ab、abc、abcc、abccc 等
ab(c)*      # 同上，并捕获字母 c
ab(?:c)*    # 同上，但不捕获字母 c
abc{2,4}    # 匹配 abcc、abccc、abcccc
(abc)*      # 匹配空字串、abc、abcabc 等
```

表3-1　正则表达式操作符优先级次序（从高到低）

优先次序	操　作　符	描　　述
最高级	() (?:)，等等	圆括号或其他分组操作符
	? + * {m,n} +? ++，等等	重复次数
	^ $ abc \G \b \B [abc]	字符序列、文字字符、字符组、断言
最低级	a\|b	多选结构

两个原子顺次排列称之为**序列**（sequence）。虽然没用标点符号，但序列也是一种操作符。为清晰起见，下文使用圆点（•）表示序列关系。上述例子因此变为：

```
a•b*•c      # 匹配 ac、abc、abbc、abbbc 等
a•b•c*      # 匹配 ab、abc、abcc、abccc 等
a•b•(c)*    # 同上，同时捕获字母 c
a•b•(?:c)*  # 同上，但不捕获字母 c
a•b•c{2,4}  # 匹配 abcc、abccc、abcccc
(a•b•c)*    # 匹配空字串、abc、abcabc 等
```

现在操作符之间的优先级关系是不是明显许多了？

在优先级次序中级别最低的，要数多选结构了。下面继续使用•记号提示：

```
e•d|j•o       # 匹配 ed 或 jo
(e•d)|(j•o)   # 同上
e•(d|j)•o     # 匹配 edo 或 ejo
e•d|j•o{1,3}  # 匹配 ed、jo、joo、jooo
```

像^或\b 这样的零宽度原子，与其他原子的优先次序是一致的：

www.TopSage.com

大家网 TopSage.com

```
^e•d|j•o$      # 匹配行首的 ed，或行尾的 jo
^(e•d|j•o)$    # 匹配仅含 ed 或仅含 jo 的行
```

优先级次序不太好记。剔除多余括号属于高级技巧，在正则表达式中尤其如此，因而特别需要留意，以防删除过多而误事：

```
# 这封邮件是谁发给我的?
/^Sender|From:\s+(.*)/;     # 错误！它可以匹配像这样的伪造头：
                            # X-Not-Really-From: faker
```

上述模式的本意是要匹配邮件头中 Sender:或 From:开头的行，但实际上能匹配到的内容显然不是我们所期望的。如果加上合适的括号，意思就清晰了：

```
/(^Sender)|(From:\s+(.*))/;
```

添加一对圆括号，或非捕获型括号(?:)（见条款 32），问题就能迎刃而解：

```
# 改良版本
/^(Sender|From):\s+(.*)/;       # 变量 $1 保存的是单词 Sender 或 From
/^(?:Sender|From):\s+(.*)/;     # 变量 $1 保存的是真正要提取的内容
```

2. 双引号变量内插

Perl 正则表达式的变量插值方式与双引号字串内的变量插值方式相同。变量名称以及\U 和\Q 这样的字符转义，并不属于正则表达式原子，因而正则表达式解析器不会处理它们。内插是一个单独的过程，发生在 Perl 解析正则表达式之前：

```
/te(st)/;      # 测试系统变量$_是否匹配 test
/\Ute(st)/;    # 匹配 TEST
/\Qte(st)/;    # 匹配 te(st)
$x = 'test';
/$x*/;         # 匹配 tes、test、testt 等
/test*/;       # 同上
```

不明白变量插值和正则表达式解析过程的先后顺序，常常会导致混淆误用的情况。请看下面的例子，要是把插值变量当成了正则表达式的原子，会发生什么情况：

```
# 读入一个模式，匹配该模式出现两次的情形
chop( $pat = <STDIN> ); # 例如，读入的模式为 bob
print "matched\n" if /$pat{2}/;     # 错误，实际相当于 /bob{2}/
print "matched\n" if /($pat){2}/;   # 正确，实际相当于 /(bob){2}/
print "matched\n" if /$pat$pat/;    # 简单罗列式，虽然相等，倒也正确
```

上例中，如果用户输入 bob，第一条正则表达式所匹配的就是 bobb，因为在解析正则表达式之前，变量 $pat 已经被实际内容替换了。

上面三条正则表达式都隐藏着另一个陷阱。假如用户输入的是字串 hello :-)，就会导致严重的运行时错误。变量内插后得到的实际正则表达式会从 /($pat){2}/ 变为 /(hello :-)){2}/，不光是毫无意义，连括号也不对称了。Perl 会报告正则表达式错在哪里：

```
Unmatched ) in regex; marked by <-- HERE in ↵
  m/(hello :-)) <-- HERE {2}/
```

这种问题我们可以用 qr// 解决（见条款 40）。

对于括号、星号、点号之类的特殊字符，如果不想把它们作为正则表达式元字符使用，可以

借助 quotemeta 操作符，或是转义操作符 \Q。quotemeta 和 \Q 会在任何不是字母、数字及下划线的字符之前加上反斜线作转义处理。

下面的代码从标准输入读取字串，使用 quotemeta 将字符中的特殊字符转义，然后再用于匹配。如果输入仍旧是 hello :-)，那么转义后会变为 hello\ \:\-\)，可以放心地用这个正则表达式进行匹配了。

```
chomp( $pat = <STDIN> );
my $quoted = quotemeta $pat;
print "matched\n" if /($quoted){2}/;
```

或者，直接在表达式中使用转义操作符 \Q 和 \E[①]：

```
chomp( $pat = <STDIN> );
print "matched\n" if /(\Q$pat\E){2}/;
```

跟使用正则表达式的其他语法结构一样，一点点疏忽都可能造成正则表达式崩溃：

```
# 它实际上相当于/hello \ \:\-\)\{2\}/，显然这是一个致命错误
print "matched\n" if /(\Q$pat){2}/; # 错误！漏掉了 \E
```

3. 要点

- 要留意正则表达式的优先级。
- 可以使用圆括号将正则表达式分组。
- 使用 \Q 或 quotemeta 将元字符转义为普通字符。

条款 29　使用正则表达式的捕获功能

虽然使用正则表达式可以非常方便地判断字串之间的模式匹配，但其作用远不止于此——它尤其适合对文本内容的分析和处理。而借助正则表达式的捕获功能，我们还可以从字串中自由提取感兴趣的部分。

1. 捕获变量：$1，$2，$3...

在使用正则表达式解析并捕获文本时，经常用到捕获变量 $1，$2，$3 等，依次类推。**捕获变量**（capture variable）与正则表达式中的圆括号相对应，有时也称为**捕获缓存**（capture buffer）。正则表达式内的每对圆括号都会“捕获”括号内匹配的文本，并将其存储到捕获变量中。比如从 URL 析取主机名和资源路径：

```
$_ = 'http://www.perl.org/index.html';
if (m#^http://([^/]+)(.*)#) {
  print "host = $1\n"; # www.perl.org
  print "path = $2\n"; # /index.html
}
```

只有匹配成功时，捕获变量才会被赋值。若是没有匹配，那么对应位置上的捕获变量不会改

① \Q 表示转义开始，\E 表示结束。Q 就是 quote，E 就是 end。——译者注

www.TopSage.com

变其值，所以可能仍然留有之前匹配时得到的内容。来看具体例子，继续上面的代码，由于匹配失败，不会更新 $1 和 $2，所以打印出来的仍然是之前匹配的结果：

```
$_ = 'ftp://ftp.uu.net/pub/';
if (m#^http://([^/]+)(.*)#) {
  print "host = $1\n"; # 仍然是 www.perl.org
  print "path = $2\n"; # 仍然是/index.html
}
```

即使圆括号内的表达式能在字串内匹配多次，它也只会捕获最后一次匹配的内容。比如下面的 $2，它可以匹配主机名后的每一段路径，但最终结束时捕获的是最后一段：

```
$_ = 'ftp://ftp.uu.net/pub/systems';
if (m#^ftp://([^/]+)(/[^/]*)+#) {
  print "host = $1\n";           # ftp.uu.net
  print "fragment = $2\n";       # /systems
}
```

要找出捕获变量数字和圆括号间的对应关系，不管括号嵌套多复杂，也只需要从左起依次数左括号的序数即可：

```
$_ = 'ftp://ftp.uu.net/pub/systems';
if (m#^ftp://([^/]+)((/[^/]*)+)#) {
  print "host = $1\n";          # ftp.uu.net
  print "path = $2\n";          # /pub/systems
  print "fragment = $3\n";      # /systems
}
```

“数左括号”法则适用于所有正则表达式，即便是多选结构中的括号也不例外：

```
$_ = 'ftp://ftp.uu.net/pub';
if (m#^((http)|(ftp)|(file)):#) {
  print "protocol = $1\n";     # ftp
  print "http     = $2\n";     # 空字串
  print "ftp      = $3\n";     # ftp
  print "file     = $4\n";     # 空字串
}
```

特殊变量 $+ 保存了最后一个非空捕获变量的内容，接续上例可以看到：

```
print "\$+ = $+\n";      # ftp, 来自 $3
```

而这种“沿袭”还不止于此！在新作用域内的捕获变量会自动本地化，唯一不同的是，本地化后变量依然沿袭外部作用域的取值，而不会像以往那样重新初始化：

```
$_ = 'ftp://ftp.uu.net/pub';
if (m#^([^:]+)://(.*)#) {
  print "\$1, \$2 = $1, $2\n";      # ftp, ftp.uu.net/pub
  {
    # 开始时，$1 和$2 的值来自外部作用域
    print "\$1, \$2 = $1, $2\n";    # ftp, ftp.uu.net/pub
    if ( $2 =~ m#([^/]+)(.*)# ) {
       print "\$1, \$2 = $1, $2\n";# ftp.uu.net, /pub
    }
  }
```

www.TopSage.com

```
    # 退出内层作用域后，$1 和$2 又恢复了原来的值
    print "\$1, \$2 = $1, $2\n";      # ftp, ftp.uu.net/pub
}
```

请注意，此处本地化使用的是 local，而非 my（见条款 43）。

2. 捕获的反向引用

正则表达式本身可以用反向引用匹配或调用之前捕获的内容，以原子 \1、\2、\3 等表示，并依次匹配相应的缓存。

关于反向引用，有一个非常典型（但未必有用）的例子，即单词重复的处理。比如，下面的表达式会找出连续重复两次的字符：

```
/(\w)\1/;
```

匹配两个或两个以上的连续重复字符：

```
/(\w)\1+/;
```

对于嵌套使用的圆括号，一定要仔细计数，以便使用正确的反向引用。比如在下面这个表达式中，因为重复两次的字符是在左边起第二个圆括号内，所以应该使用 \2 的形式表示反向引用：

```
/((\w)\2){2,}/;
```

我们还可以重复使用反向引用。比如下面的例子就是用来搜寻同一个元音字符重复出现四次的表达式。第一行比较繁琐，但便于观察重复出现的反向引用模式；第二行的表达方式更自然，可以直接计数：

```
/([aeiou]).*\1.*\1.*\1/;
/([aeiou])(.*\1){3}/;
```

或许你还无法体会反向引用的妙处和用场，但这个特性真的非常强劲。比如处理定界符的识别和匹配，有了反向引用，写出来的正则表达式可以非常简洁：

```
q/"stuff"/ =~ /(['"]).*\1/;
```

也许贪婪匹配并不合适，万一吃掉太多引号的话结果多半不对。所以恰当的写法是用懒惰匹配（见条款 34）：

```
q/"stuff","more"/ =~ /(['"]).*?\1/;
```

你还可以用这种更酷的写法，顺便解决转义引号的误判问题：

```
q/"stuff\"more/ =~ /(['"])(\\\1|.)*?\1/;
```

不过，这种方法也有缺点：无法用来匹配成对的圆括号（即便不考虑嵌套括号的情形），而对于原本写成转义形式的字符，我们也有许多其他又快又好的处理办法。

3. 捕获并替换

捕获及匹配变量常常用于替换操作。在替换字串中出现的 $1、$2、$& 等变量，指的是在当前匹配区内捕获的缓存，它们不受先前赋值的影响，仅限当前正则表达式内有效。所以知道了这一点，就能理解下面交换两个单词位置的写法了：

www.TopSage.com

```
s/(\S+)\s+(\S+)/$2 $1/;
```

对于简单的HTML标签转换，可以通过散列表查询，取得要转换的目标形式：

```
my %ent = { '&' => 'amp', '<' => 'lt', '>' => 'gt' };
$html =~ s/([&<>])/&$ent{$1};/g;
```

利用替换操作还可以生成缩略词。比如取新闻组名称的首字母：

```
$newsgroup =~ s/(\w)\w*/$1/g;
```

要是替换的目的是去除匹配的内容，而非保留改写，那就不要使用捕获，用了也是浪费。请看下面两行代码，一样是删除行首空格，第一行又是贪婪又是捕获，第二行则干脆利落直接删除，既快又好：

```
s/^\s*(.*)/$1/;    # 显然是浪费

s/^\s+//;          # 好多了
```

在需要计算替换结果的情况下，我们可以用 /e (eval)[①]修饰，这是一种解决运行时判决替换结果的技巧。比如，对于所有不存在的文件，在其文件名后追加 not found 的信息：

```
s/(\S+\.txt)\b/-e $1 ? $1 : "<$1 not found>"/ge;
```

使用 /e 作替换时，我们常常结合 /x 修饰及成对的正则表达式界定符一起使用，以便增加程序的可读性（见条款37）：

```
s{
  (\S+\.txt)\b # 所有以.txt 结尾的文件名
}{
  -e $1 ? $1 : '<$1 not found>'
}gex;
```

4. 列表上下文中的匹配

在列表上下文中，匹配操作会根据所捕获的缓存，返回与其对应的列表。如果匹配失败，则返回空列表，但此时捕获变量 $1、$2、$3 等不会受其影响，依然保留原来的值。

这应该是匹配符最常用的特性之一了。只需一步便能扫描并同时分割字串。下面这条依据RFC 2822规范析取邮件头信息的语句，就是利用这一特性写得非常简单紧凑：

```
my ( $name, $value ) = /^([^:]*):\s*(.*)/;
```

你可以有选择性地提取合适的内容。比如下面的代码，返回的邮件标题将不会包含回复邮件时加上的前缀 Re:：

```
my ( $bs, $subject ) = /^subject:\s+(re:\s*)?(.*)/i;
```

由于我们并不需要标题的前缀，所以利用列表切片，可以只返回标题内容：

```
my $subject = (/^subject:\s+(re:\s*)?(.*)/i)[1];
```

或者，直接省掉捕获内容的缓存，对 Re: 部分使用非捕获括号：

```
my ($subject) = /^subject:\s+(?:re:\s*)?(.*)/i
```

① 相当于 eval 语句，替换区的内容将作为 Perl 代码求值计算，替换时使用计算结果。——译者注

www.TopSage.com

TopSage.com

结合 map 循环返回匹配结果的写法，可以使代码更为流畅简洁。比如下面这行提取邮件发送日期的代码，从邮件头数组到最后的日期内容，一气呵成：

```
my ($date) = map { /^Date:\s+(.*)/ } @msg_hdr;
```

我们知道，失败的匹配会返回空列表，而正是由于这种特性，上面的写法可以忽略所有非日期行的邮件头，而将找到的第一行能匹配的日期赋于$date 变量。

5. 用正则表达式切词

捕获功能还有一项比较有趣的应用，就是对字符串**切词**（tokenize）——利用空白字符、数字、标识符、操作符等诸如此类的词法元素，将字符串切分为一系列单词的操作。

如果你用 Perl 写过（或尝试过）计算机语言的语法解析工具，可能就会觉得 Perl 好像缺少了什么理应提供的功能，使得解析起来困难重重。而事实上，这的确非常难。问题在于解析字符串时，要确定的是如何根据字符串开头，推算出众多可能模式中最为合理的一种。而 Perl 擅长的，却是以固定模式测试字符串。这两者确实很难搭配到一起。

来看一个解析数学算式的例子。算式应该包含数字、括号以及 +、-、*、/ 等算符。这里我们不考虑空白字符的影响，假设解析之前已经先行去掉。

最初看似直观的做法，是枚举各种可能的情况，然后将捕获到的数字或算符依次压入数组：

```
my @tokens;

while ($_) {
  if (/^(\d+)/) {
    push @tokens, 'num', $1;
  }
  elsif (/^([+\-\/*()])/) {
    push @tokens, 'punct', $1;
  }
  elsif (/^([\d\D])/) {
    die "invalid char $1 in input";
  }
  $_ = substr( $_, length $1 );
}
```

虽然代码不够美观，但效率上勉强还能接受。因为这里全部使用了以定位锚^开始的正则表达式，所以只在开头尝试匹配，不会逐个位置一一测试，运行起来相当快。

但如果解析较长字串的话，上面代码的效率会大幅下降，问题出在末尾的 substr 操作。或许你会想到一个解决办法，将匹配字串最后所在位置当作字串长度，保存到变量 $pos 中，节约 length 函数的计算开销：

```
if ( substr( $_, $pos ) =~ /^(\d+)/ ) { ... }
```

可惜，这么做并不会快很多；并且对较短字串而言，速度反而可能会下降。

其实有一种方法不但效率不错，而且还不受匹配字符的长短影响，它利用 /g 修饰在标量上下文中的返回结果，以判断字符串是否属于某种特定的计算模式。每次执行 /g 匹配，正则表达式引擎都会从上次成功匹配的结束位置开始寻找新的匹配。这就可以利用单条正则表达式匹配某

种算式模式，从而免去手工维护匹配位置的麻烦：

```
# 使用 m//g 的方法
while (
    /
    (\d+)          |    # 数字
    ([+\-\/*()]) |    # 运算符
    (\D)               # 数字以外的字符
    /xg
  )
{
  if ( $1 ne "" ) {
    push @tok, 'num', $1;
  }
  elsif ( $2 ne "" ) {
    push @tok, 'punct', $2;
  }
  else {
    die "invalid char $3 in input";
  }
}
```

6. 要点

- 仅在匹配成功时使用捕获变量。
- 在列表上下文中使用匹配操作符，就能以列表的形式保存匹配结果。

条款30　使用更精确的空白字符组

空白字符这个话题乍看上去很简单，实际上却隐藏着不少陷阱。空白字符包括水平空白字符，如英语单词的间隔，以及各种排版用的为便于阅读而增补的空白字符，等等。

而与此类似的，还有一种垂直空白字符，一般就是指换行符。其实，计算机才不管是水平还是垂直，对它来说，所有的内容都只是字符而已，并无高低上下之分。只不过为方便起见，程序特意指定某个字符作为分隔符，以便将文本划分为更小的单位罢了。

Perl的正则表达式提供了各种精确匹配空白字符的方法。

1. 水平空白字符

在 Perl 5.10 之前，预先定义好的有关空白字符处理的只有两组。一组是 \s，匹配所有类型的空白字符，另一组是 \S，匹配除空白字符外的任意字符。使用这种一刀切式的分类，稍有不慎就会导致意料之外的结果。比如下面这段代码，为了把连续几个空白字符替换成单个空格，第一直觉一定会想到用 \s+ 匹配：

```
my $string = <<'HERE';
This   is     a        line
This     is     another line
And a       final   line
HERE
```

www.TopSage.com

```
$string =~ s/\s+/ /g;
```

由于换行符也属于空白字符，所以上述代码会将换行符也替换成空格，结果变成了单行文本：

```
This is a line This is another line And a final line
```

为了避免这种情况，你也许已经想到必须明确指定要替换的字符集合：

```
$string =~ s/[ \t]+/ /g;
```

你可能会觉得这样替换掉所有空格和跳格符，应该就没什么问题了吧。可是，Unicode 字符集中其他形式的白字符又该如何处理呢？显然，上面这种非常局限的写法不能解决问题。

从 Perl 5.10 开始，我们就可以用 \h 字符组匹配任意水平空白字符：

```
use 5.010;

$string =~ s/\h+/ /g;
```

现在所有连续的水平空白字符都将被替换为单个空格，而换行符被保留下来。

正如 Perl 其他字符组一样，\h 也有其相反表示，即大写的 \H，它会匹配所有水平空白字符以外的任意字符，包括垂直空白字符以及所有非空白字符。

2. 垂直空白字符

和上面只需处理水平空白字符的情景类似，你或许会遇到只需处理垂直空白字符的情况。垂直空白字符可以分为回车符、换行符、进纸符（form feed）、垂直制表符，以及 Unicode 的行分隔符和段分隔符，等等。比如，想要看看某段文本是否为多行文本，可以用 \v 匹配任意垂直空白字符[①]：

```
use 5.010;

if ( $string =~ m/\v/ ) {
  say 'Found a multiline string';
}
```

如果要把多行文本按行切分，也可直接在 split 中使用 \v：

```
my @lines = split /\v/, $multi_line_string;
```

和 \v 对应的是大写的 \V，它匹配垂直空白字符以外的任意字符，包括水平空白字符以及所有非空白字符。

3. 行终止符

长期以来，对于行终止符的处理，一直很麻烦。因为，不但有换行符，还有回车符；其组合方式也不一样，有的文件用换行符结束当前行，有的文件用回车符再加上换行符的形式表示换行。事实上，许多网络协议也是采用回车符换行符组合的方式，而 Unicode 又引入了某些新的换行符。

① 进纸符简记为\f，对应 ASCII 的<FF>字符；换行符简记为\n，对应 ASCII 的<LF>字符；回车符简记为\r，对应 ASCII 的<CR>字符。——译者注

www.TopSage.com

要是你的任务和行终止符相关，就需要同时处理所有可能的行终止符情况。一般你可能会将一个可选的回车符后跟一个换行符的形式，替换为普通的换行符：

```
$string =~ s/(?:\r?\n)/\n/g;
```

然而这种方式的效果并不好，因为 \r 和 \n 都属于逻辑字符，但在不同操作系统上对应的字符编码并不完全一致（Mac Classic，说的就是你!）。你或许会想到用字符组的形式，不管这些字符怎么组合出现几个，都能一次匹配：

```
$string =~ s/[\r\n]{1,2}/\n/g;
```

不过，这也可能会将\n\n 替换为\n，即两个连续的空行会变为一行。更何况，它对 Unicode 行终止符一样无能为力。

为了处理所有种类的行终止符，我们可以写一条复杂但完备的正则表达式，它包含了各种原本不为所知的行终止符，像垂直制表符 0x0B 和进纸符 0x0C：

```
$string =~
  s/(?>\x0D\x0A?|[\x0A-\x0C\x85\x{2028}\x{2029}])/\n/g;
```

为了免去这种麻烦，Perl 5.10 特意引入的 \R 字符组，以更简洁的方式表示，但作用则完全相同：

```
use 5.010;

$string =~ s/\R/\n/g;
```

现在，所有形式的行终止符都变成简单统一的换行符了。

4. 非换行符

Perl 5.12 还引入了一种新的用来表示非换行符的字符组 \N。之前，我们可以用 . 匹配任意非换行符的字符，例如：

```
if ( $string =~ m/(.+)/ ) { ... }
```

如果有人添加了 /s 修饰，这或许是因为不明就里地遵循了“最佳实践”那本书中推荐的做法，使得 . 符号突然就能匹配换行符了，从而破坏程序原来的逻辑：

```
if ( $string =~ m/(.+)/xsm ) { ... } # 乱了!
```

如果我们只是需要匹配非换行符，那现在可以显式使用 \N 匹配，不必担心 /s 来捣乱：

```
use 5.012;

if ( $string =~ m/(\N+)/ ) { ... }
```

同字符组 \N 相反的，并不是对应字符组的补集，而仅仅是一个换行符 \n。下次参加 Perl 掌故大赛（Perl Trivia Challenge）时，可别忘了这点哦。

5. 要点

- \h 匹配水平空白字符。
- \v 匹配垂直空白字符。
- \N 匹配非换行符。

www.TopSage.com

条款31 使用命名捕获，给匹配加标签

有时正则表达式中出现的括号太多，记忆变量编号与圆括号间的对应关系相当麻烦，要是能免掉就好了。好吧，先来看看究竟有哪些麻烦，下面的代码用于匹配人名：

```
$_ = 'Buster and Mimi';
if (/(\S+) and (\S+)/) {
  my ( $first, $second ) = ( $1, $2 );
  ...;
}
```

程序完成后如果需求变更，可能会加上更多括号，但难免会出现忘记一并修改捕获编号的情况。这可能发生在：

```
$_ = 'Buster or Mimi';
if (/(\S+) (and|or) (\S+)/) {
  my ( $first, $second ) = ( $1, $2 ); # 糟糕!
  ...;
}
```

设想一下，要是添加的括号不止一个，情况会怎样？

如果不用记忆编号和捕获的对应关系，那该多方便。如果你用的是Perl 5.10或更高版本，那么恭喜你，你可以使用命名捕获语法来应付这种状况。它的形式只比原来复杂一点点：(?<LABEL>)[①]。而匹配之后，捕获的内容会保存在散列表%+中，以尖括号内的名字作为散列表的键名：

```
use 5.010;

$_ = 'Buster and Mimi';
if (/(?<first>\S+) and (?<second>\S+)/) {
  my ( $first, $second ) = ( $+{first}, $+{second} );
  ...;
}
```

现在，即便再加入其他圆括号，也不会破坏程序原来的逻辑，因为匹配内容的获取和圆括号顺序完全无关：

```
use 5.010;

$_ = 'Buster and Mimi';
if (/(?<first>\S+) (and|or) (?<second>\S+)/) {
  my ( $first, $second ) = ( $+{first}, $+{second} );
  ...;
}
```

这种技术同样适用于反向引用。以前需用\1、\2等一一编号，现在只用\k<label>语法引用命名捕获即可：

① 这种语法很好记，每一个部分都不可缺少：尖括号限定标签名，问号是在捕获内加注其他语法的统一标记。去掉任何一个元素都会造成歧义。——译者注

www.TopSage.com

大家网 TopSage.com

```
use 5.010;

$_ = 'Buster and Buster';
if (/(?<first>\S+) (and|or) \k<first>/) {
  say 'I found the same name twice!';
}
```

虽然我们对某些捕获加上了标签，但 Perl 内部依然还会使用编号变量追踪，所以匹配之后究竟用哪种形式提取捕获内容，应视具体情况而定。

Perl 5.10 还引入了相对位置的反向引用，这样就不必记住每一组捕获的绝对位置。以绝对位置定位反向引用的写法是 \g 加数字编号：

```
if (/(?<first>\S+) (and|or) \g1/) {
  say 'I found the same name twice!';
}
```

这种写法必须对每一组捕获的编号从头数起。如果我们需反向引用的部分是在末端，则倒过来数更方便，而且不用担心前面圆括号的增减，这种方式我们称为相对反向引用。举个例子，表示倒数第二个就用 -2，但这个负数两边得加上花括号：

```
if (/(?<first>\S+) (and|or) \g{-2}/) {
  say 'I found the same name twice!';
}
```

要点

- 从 Perl 5.10 起，可以使用命名捕获。
- 命名捕获的结果保存在散列表 %+ 中。
- 使用相对反向引用，可免去从头计算捕获编号的麻烦。

条款 32　仅需分组时，用非捕获括号

圆括号在 Perl 正则表达式中，有两种截然不同的作用：分组和捕获。一般来说，这两种功能并在一起使用还是挺方便的，大抵也能相安无事，不过偶尔也会出现问题。比如之前看到的例子，匹配邮件标题行 Subject:，忽略可能出现的回复前缀，直白地写出来会用到两组圆括号：

```
my ( $bs, $subject ) = /^subject:\s+(re:\s*)?(.*)/i
```

第一对圆括号的作用是分组（使后面的 ? 能正常工作），但再将内容捕获就显得多余。这里，我们需要的是没有捕获的分组。

Perl 为此提供了专门的解决方案。非捕获括号 (?:) 的用法与普通圆括号相同，唯一区别在于，它不会创建反向引用或捕获变量。如果只是需要分组，那就没必要额外复制一份圆括号内匹配到的内容。而且，使用 Perl 的非捕获括号，还可以省下复制所用的时间：

```
my ($subject) = /^subject:\s+(?:re:\s*)?(.*)/i;
```

有的匹配模式使用嵌套括号，比如在匹配主机名的表达式 (\w+(\.\w+)*) 中，我们根本不关心内层圆括号匹配的内容，因此，用非捕获括号可以稍微节省点时间：

```
my ($host) = m/(\w+(?:\.\w+)*)/;
```

其实，除非是大段文本，否则这也省不了多少时间，而且非捕获括号也不会对程序的可读性有太多提升。但有时，即使一丁点儿的速度提升都至关重要。究竟能节省多少时间，可以测试一下正则表达式的性能（见条款 41）。

map 循环中的正则匹配经常会用到非捕获括号，因为是在列表上下文中，所以最终从 map 返回的，仅仅是成功匹配的内容：

```
my @subjects =
  map { /^subject:\s+(?:re:\s*)?(.*)/i } @headers;
```


你还可以在 split 中使用非捕获括号，以便禁用**分隔符保留模式**（separator-retention mode）。一般情况下，在 split 中捕获的分隔符，也会返回到输出的列表中：

```
my $string = '1:2:3:4';
my @items = split /(:)/, $string;
```

这就是分隔符保留模式，返回的列表中包含每一个分隔符。像上面的 @items，其内容由列表 qw(1 : 2 : 3 : 4) 组成，包括数字和冒号。如果原本想要知道各元素间的分隔符是什么，这种写法不错，但多数情况下，我们并不需要分隔符。

如果分隔符是：或者；，并且分隔符两边有空白字符，那么写正则表达式的时候，必须将，|；放在圆括号内，以便去除两边的空白字符：

```
my $string = '1:2; 3: 4 ;5';
my @items = split /\s*(,|;)\s*/, $string;
```

你原本没想要保留分隔符，但它们可能会不请自来。而借助非捕获括号，就没有这样的问题：

```
my @items = split /\s*(?:,|;)\s*/, $string;
```

要点

- 如果仅需分组功能，请使用非捕获括号。
- 在 split 中使用非捕获括号，以关闭分隔符保留模式。

条款 33　小心处理匹配变量

匹配变量（$`、$& 以及 $'）[①]会让 Perl 程序的执行速度大打折扣，因为它需要额外的时间和空间保存信息。每次成功匹配之后，这三个匹配变量分别保存着匹配内容之前的文本、匹配内容本身以及匹配内容之后的文本：

```
'My cat is Buster Bean' =~ m/Buster/;

print <<"HERE";
 Prematch: $`
```

① 可以将`和'想象为一对引号，中间的部分用$&表示，之前的部分用“左引号”$`表示，之后的部分用“右引号”$'表示。这样就容易记住，而且不会轻易遗忘。Perl 里面的标点符号看似杂乱，实则有章可循。Everything happens for a reason。——译者注

www.TopSage.com

```
    match: $&
Postmatch: $'
HERE
```

输出结果显示了匹配内容以及环绕它的文本：

```
 Prematch: My cat is
    match: Buster
Postmatch:  Bean
```

匹配变量可以在替换操作时用于处理文本。比如将 HTML 中的某些注释替换为特定数据：

```
# 替换某些 html 注释
while (<OLD>) {
  if ( /<!--\s*(.*?)\s*-->/ and ok_to_replace($1) ) {
    $_ = $` . html_data($1) . $';
  }
  print NEW $_;
}
```

1. 性能问题

有人抱怨使用匹配变量会导致 Perl 程序更加迟缓。这是真的——确实有人为此抱怨。

不过，匹配变量的确会影响效率。对某次匹配来说，它们确实能派上用场，但在随后的正则表达式中，只要遇到这类变量，不管是否真的需要记录匹配内容，Perl 都会一一记录。这部分额外的工作会明显拖慢程序的运行：

```
'My cat is Buster Bean' =~ m/Buster/;
print "I matched $&\n";

while (<>) {
  next unless /Bean/; # 每次匹配都要付出代价!
}
```

即使你没打算用这些匹配变量，也可能会遇到它们。比如 English 模块为 $`、$& 和 $' 变量提供了等价的长名称变量。只要在程序中加载这个模块就会触发上述问题：

```
use English; # 嗯，现在全慢下来了
```

变通的办法是告诉这个模块，不要为匹配变量引入这些变量：

```
use English qw(-no_match_vars);
```

2. 使用 /p 修饰

从 Perl 5.10 起引入了新的正则表达式修饰符 /p，它只对当前匹配提供匹配变量。启用后，所匹配的内容会分别保存在变量 ${^PREMATCH}、${^MATCH} 和 ${^POSTMATCH} 中，因而，不会影响其他正则表达式的行为：

```
use 5.010;

'My cat is Buster Bean' =~ m/\s\w+\sBean/p;
say "I matched ${^MATCH}";

while (<>) {
```

www.TopSage.com

```
    next unless /Bean/; # 速度不受影响!
}
```

3. 要点

- 避免使用匹配变量，它多少会影响效率。
- 使用 English 模块时要小心，以免误用匹配变量。
- 在 Perl 5.10 中，可以用 /p 修饰启用单次生效的匹配变量。

条款 34　能懒则懒，不要贪婪

此处**贪婪**（greedy），与钱无关，我们只是在说正则表达式匹配时的一种行为方式，它是一个术语，对大多数正则表达式引擎来说都适用，当然也包括 Perl。

一般的规则是，Perl 正则表达式默认总会返回它所能找到的“最左最长”（Leftmost Longest）匹配，即在字串中找出第一个能匹配且尽可能长的字串。像 * 和 + 这样表示重复次数的操作符，会“吃进”尽可能多的字符，这就是我们所说的贪婪：

```
$_ = "Greetings, planet Earth!\n";
/\w+/;     # 匹配 Greetings
/\w*/;     # 匹配 Greetings
/n[et]*/; # 匹配 Greetings 中的 n
/n[et]+/; # 匹配 planet 中的 net
/G.*t/;    # 匹配 Greetings, planet Eart
```

一般我们总是希望以贪婪模式匹配，不过也有例外。要是你只希望匹配两个单引号之间的文本呢？请看下面的例子，显然在贪婪模式作用下匹配了太多的东西：

```
# 要匹配单引号之间的内容——并非如你所愿!
$_ = "This 'test' isn't successful?";
my ($str) = /('.*')/; # 实际匹配到的是 test' isn
```

匹配结束的条件已经满足，Perl 却不停止继续匹配，是这样吗？当然不是。模式中的点号 . 确实匹配了单引号 '，只不过到最后一次出现 ' 的位置才结束匹配。

在上例中，你可以使用排除法来匹配单引号以外的字符，以便消除点号带来的副作用：

```
# 用传统方式匹配单引号之间的字串
$_ = "This 'test' isn't successful?";
my ($str) = /('[^']*')/; # 匹配的内容是 test
```

幸运的是，Perl 还提供了懒惰模式的重复操作符。懒惰模式非常强大，也极为有用，我们可以用它写出简单的正则表达式，巧妙地解决某些难题，而这些难题原本可能是非常复杂，甚至令人束手无策的。

对于任意重复操作符（*，+，{m,n}），在后面加上问号（?）就会变为**非贪婪**（non-greedy）模式，也称懒惰模式。它只匹配尽可能短的字串，一旦满足匹配，立即结束。用这种方式解决上面的问题，言简意赅：

```
# 用懒惰模式匹配单引号之间的内容
$_ = "This 'test' isn't successful?";
```

www.TopSage.com

```
my ($str) = /('.*?')/; # 匹配的内容同样是 test
```

现在我们来尝试下更有难度的操作，比如匹配带有转义字符的双引号字串（例如 \"、\\ 和 \123）：

```
# 匹配双引号内的字串
$_ = 'a "double-quoted \"string\042"';
my ($str) = /("(\\["\\]|\\\d{1,3}|.)*?")/;
print $str; # 得到 double-quoted \string\042
```

懒惰模式唯一的问题在于可能会导致效率降低。所以，如非必需，请不要随意使用懒惰模式。但如果能避免编写复杂或有问题的正则表达式，就没什么可犹豫的，毕竟，解决问题为先。

要点

- Perl 中表示匹配次数的量词，会默认匹配尽可能长的内容。
- 避免使用超出需要的匹配模式。
- 在量词后加上 ?，转为懒惰模式。

条款35　用零宽断言匹配字串中的特定位置

有时我们会遇到仅需匹配某种条件，而非某个字符的情况。比如定位单词边界，字串起始、结束位置，或是字串中文本行的起始、结束位置。Perl 的正则表达式提供了**定位锚符**（anchor），以匹配上述位置。这些定位锚符也称为**零宽断言**（zero-width assertion），因为它们只是用作条件判断，不会匹配任何字符。

1. 用\b 判断单词边界

请看下面的正则表达式，从命令 who 输出的内容析取用户名和终端名称：

```
# 刚开始是正常的
my @who = `who`;
$_ = pop @who;
my ( $user, $tty ) = /(\w+)\s+(\w+)/;
```

对于 joebloe ttyp0 ...这样的输入，上面的程序是可以正常工作的。不过，对于 webmaster-1 ttyp1 ...这样的字串，它就无法匹配了；而对于 joebloe pts/10，它的返回值可能会很奇怪。如果把单词匹配改成非空白字符，就不会有类似问题了：

```
my ( $user, $tty ) = /(\S+)\s+(\S+)/;
```

在写正则表达式时，如果将 \w 与 \s 或者 \W 与 \S 连用，就很有可能会出问题。如果一定要用，至少应该先仔细检查。

另外需要注意的是，某些英语“单词”中可能包含标点符号。比如在一段英文文本中搜索某个完整的英文单词：

```
print "Enter a word to search for: ";
my $word = <STDIN>;
print "found\n" if $text =~ /\b\Q$word\E\b/;
```

www.TopSage.com

它可以处理 hacker 或者 Perl5-Porter 这样的输入内容，但却无法处理 goin'。推而广之，它无法处理任何不是以 \w 开始和结尾的单词。如果 $text 中含有 isn't，程序还将 isn 视为可匹配的单词。而 \b 匹配的是 \w 和 \W 之间相互转换时的位置，不是 \s 和 \S 字符之间的位置。想要以空白字符作为分隔符提取单词，只能改写成这样：

```
print "Enter a word to search for: ";
my $word = <STDIN>;
print "found\n" if $text =~ /(^|\s)\Q$word\E($|\s)/;
```

单词边界锚点 \b，及其反义形式的 \B，都是零宽断言。它们不是唯一的零宽断言（另外还有 ^、\A 等），但却是最容易用错的。如果不确定 \b 和 \B 会在何处匹配，试着用替换操作将这些地方标注出来：

```
my $text = q(What's a "word" boundary?);
( my $btext = $text ) =~ s/\b/:/g;
( my $Btext = $text ) =~ s/\B/:/g;
print "$btext\n$Btext\n";
```

运行该程序，出现冒号的地方会被 Perl 视为单词边界：

```
% tryme
:What:':s:  :a:  ":word:"  :boundary:?
W:h:a:t's  a  :"w:o:r:d":  b:o:u:n:d:a:r:y?:
```

你可能会注意到字串首尾位置的处理有些微妙。请注意，字串的开始与结尾位置都是虚拟的非单词字符。如果字串的最后一个（或第一个）字符不是 \w 字符，该位置就不会涉及单词边界的问题。同时还要注意，连续的 \W 字符之间（例如空格与双引号），不存在单词边界，而连续的 \w 字符之间也是如此。

2. 用 ^ 或 \A 匹配起始位置

^ 通常用来匹配字符串的起始位置。许多人误以为这是“行起始”锚点，是因为他们只用它处理过单行文本。请看在下面的例子中，最终匹配的仅仅是第一个单词，即便加了 /g 修饰也是如此：

```
my $string = <<'HERE';
This is a line
That is another line
And a final line
HERE

my (@matches) = $string =~ m/^(\w+)/g; # 只匹配 This

print "@matches\n";
```

输出结果显示只匹配了第一个单词：

```
This
```

如果要匹配每行的第一个单词，可以用 /m 修饰打开**多行模式**（multiline mode）。/m 会改变 ^ 的行为，使它不但匹配字串起始，还能匹配换行符后的位置，即各行行首：

```
my (@matches) = $string =~ m/^(\w+)/mg;
```

现在的输出结果是每行起始位置的单词：

```
This That And
```

多行模式下，如果只想匹配整个字串的起始位置，可以用 \A 定位。请看下面的代码，该正则表达式会匹配字串开头位置以 T 开始的单词，以及其他行行首位置首字母不是 T 的单词：

```
my (@matches) = $string =~ m/
    (           # $1 的开始
        (?:     # 非捕获组
        \A T    # 字串开始位置，之后是 T
            |   # -或-
        ^ [^T]  # 行首位置，但单词首字母不是 T
        )
    \w+         # 单词的剩余部分
    )           # $1 到此结束
    /xmg;
```

结果表明只匹配了两个单词，而且是每种定位锚符各匹配了一个：

```
This And
```

利用行首定位锚符还可以让某些任务完成得更漂亮。比如从 here 文档读取每行两个字段，并一步初始化%scores 散列：

```
my %scores = <<'EOF' =~ /^(.*?):\s*(.*)/mg;
fred: 205
barney: 195
dino: 30
EOF
```

3. 用 $ 或\z 匹配结束位置

$通常用于匹配字串结束位置，即便此处有换行符也能匹配：

```
if ( "some text\n" =~ /text$/ ) {
  print "Matched 'text'\n";
}
```

这就好像换行符根本不存在一样。如果删去字串里的换行符，它依然能匹配：

```
if ( "some text" =~ /text$/ ) {
  print "Matched 'text'\n";
}
```

但若需要匹配字串的绝对结束位置，得在程序中明确指出，利用否定前瞻模式（negative lookahead）确保该位置是绝对结尾，而不是字串中间的换行位置：

```
if ( "some text\n" =~ /text(?!\n)$/ ) { # 失败
  print "Matched 'text'\n";
}
```

这里 (?!\n)[①]的效果是确保 $ 之后没有换行符，这就足以迫使$匹配真正的字串结尾。

① 圆括号表示位置，问号表示后面要追加另外的语法，追加的是感叹号，一般用作否定意义，表示不是后面那个字符。连起来就是说，text 之后的那个位置上不能是\n，而是行尾，才能匹配。——译者注

www.TopSage.com

其实大可不必如此劳神费力，Perl有个专门用于匹配字串末位的锚位操作符\z。而下面的正则表达式匹配失败是由于text之后有换行符，即text并非位于字串的绝对结尾位置：

```
if ( "some text\n" =~ /text\z/ ) { # 失败
  print "Matched 'text'\n";
}
```

对应于严格限定的\z，大写的\Z锚位表示一般意义上的字串结尾，它和$的行为一致，能匹配出现换行符的情况：

```
if ( "some text\n" =~ /text\Z/m ) { # 再次匹配成功
  print "Matched 'text'\n";
}
```

若是加上/m修饰，$能匹配多行字串内换行符之前的位置：

```
print "fred\nquit\ndoor\n" =~ /(..)$/mg;
```

上下两例的输出结果都是editor，即每行文本的最后两个字母：

```
print "fred\nquit\ndoor" =~ /(..)$/mg; # 结果同上
```

4. 要点

- 用\b匹配单词与非单词之间的边界。
- 用\A或\z分别匹配字串的绝对起始或绝对末尾。
- 在多行模式下，用^或$分别匹配每一行文本的行首或行尾。

条款36 简单字串处理应避免使用正则表达式

正则表达式用起来很顺手，但对于各种字符串操作，正则表达式并不总是最高效的手段。像提取字串子串、转换字符之类的简单任务，正则表达式处理起来当然不在话下，但它更适合处理更复杂的任务。在Perl中，简单的字串操作应该使用特定的函数，比如index、rindex、substr以及tr///等。

请记住，所有正则表达式匹配过程都会涉及匹配变量的处理，即使是最简单的正则表达式也不例外。如果你需要的只是对字串进行相等比较，或是提取子串，那么很明显处理匹配变量就是浪费时间。因此，只要不涉及特殊情况，应该尽可能使用内置的特定函数处理，而非事事求助于正则表达式。

1. 字串比较操作符

如果要比较两个字串是否相同，可以用字串比较操作符，而非正则表达式：

```
# 较快的方法
if ( $answer eq 'yes' ) { something_wonderful() }
```

使用字串比较操作至少比使用正则表达式快两倍。如果不用定位锚符的话，性能差距更大：

```
# 较慢的方法
if ( $answer =~ /^yes$/ ) { something_wonderful() }
```

www.TopSage.com

```
# 不用定位锚符则更慢（而且未必正确）
if ( $answer =~ /yes/ ) { something_wonderful() }
```

即便涉及某些更为复杂的比较，字串比较操作也会快些。比如忽略大小写对字串作比较：

```
# 较快
if ( lc($answer) eq 'yes' ) { something_wonderful() }

# 较慢
if ( $answer =~ /^yes$/i ) { something_wonderful() }
```

2. 用 index 和 rindex 提取子串

index 操作符可以在较长字串中定位一个较短子串的位置。而 rindex 则是从最右端起定位，但位置的计算方式仍是从左到右：

```
# 在$big_str 里寻找$little_str 的位置，起始下标是 0
my $pos = index $big_str, $little_str;

# 从$big_str 右边起，第一个出现$little_str 字串的起始位置
my $pos = rindex $big_str, $little_str;
```

index 操作符的运算速度非常快，因为搜索时它使用的是 Boyer-Moore 算法[①]。Perl 会把类似于 index 的正则表达式编译为 Boyer-Moore 搜索模式。你可以用一个匹配操作符，取得 $' 的长度，即匹配位置之前的文本长度（尽管匹配操作会比较慢）：

```
$big_str =~ /\Q$little_str/; # 千万别这样写
my $pos = length $';

# 虽然用了 pos 函数，但仍然很慢
$big_str =~ /\Q$little_str/g; # 或者可以用/og
my $pos = pos($big_str) - length($big_str);
```

用正则表达式匹配字串需要付出较高的代价。与之相比，使用 index 的速度要快许多倍，就算正则表达式用了 /o 编译选项也难改变这种状态。

3. 用 substr 提取或修改子串

substr 操作符的作用是根据给定起始位置和字串长度提取字符串中的一部分内容。如果没有指定长度，则提取从指定位置开始一直到字串末尾的字串：

```
# 得到"Perl"
my $perl = substr "It's a Perl World", 7, 4;

# 得到"Perl World"
my $perl_world = substr "It's a Perl World", 7;
```

substr 比等效的正则表达式快多了：

```
# 较慢的做法
my ($perl) = ( "It's a Perl World" =~ /^.{7}(.{4})/ );
```

① Boyer-Moore 算法，简称 BM 算法，详见 http://en.wikipedia.org/wiki/Boyer–Moore_string_search_algorithm。——译者注

www.TopSage.com

如果把 substr 表达式放在赋值操作的左侧，就是字串替换操作，即右侧字串将替换由 substr 指定位置和长度的字串，这种写法非常优雅：

```
# 修改字串变量 world
my $world = "It's a Perl World";
substr( $world, 7, 4 ) = "Mad Mad Mad Mad";
```

将 index 和 substr 结合起来，还可以实现类似于 s/// 替换的功能。不过如果目的只是替换的话，倒不如还是用 s///，不但更快，而且意义明晰：

```
# 搜索 Perl 字眼，然后替换掉
substr( $world, index( $world, Perl ), 4 ) =
  "Mad Mad Mad Mad";

# 代码更清晰，速度也可能更快
$world =~ s/Perl/Mad Mad Mad Mad Mad/;
```

4. 批量转换单个字符

如果想把某个字符一次性全部转换为另一个，也没必要使用正则表达式替换。牵一发而动全身，不但整个正则表达式引擎都得启动，而且副作用也会随之而来：

```
$string =~ s/a/b/g;
```

不要忘了 Perl 还有 tr///操作符，即字符转换操作符。虽然形式相近，但它不是正则表达式，它所做的只是字符间的转换而已。tr/// 没有 /g 选项，因为默认就是全局转换：

```
$string =~ tr/a/b/;
```

字符转换操作会将两边的对应字符一一替换掉。你还可以分别指定字符范围。下面是 Perl 版的 ROT-13 编码算法[①]，将每个字符循环替换为字母表中 13 位之后的字符：

```
$string =~ tr/a-mA-Mn-zN-Z/n-zN-Za-mA-M/;
```

如果只想对字串某一部分作字符转换，可以用 substr 限定范围：

```
substr( $string, 0, 10 ) =~ tr/a/b/;
```

5. 要点

- 不要一遇到问题就用正则表达式解决。
- Perl 自带丰富的字串函数，应物尽其用。
- 用字符转换操作符批量转换单个字符。

条款 37　提高正则表达式的可读性

通常来说，正则表达式是“天书”的代名词。无可否认，正则表达式尽管简洁紧凑，却易写难读，它自成体系，是一门微型的编程语言，但它不含 foreach 或 while 之类的关键词，而是用 \w、[a-z] 和 + 之类的原子。

① Rot-13 编码算法，即回转 13 位算法，详见 http://zh.wikipedia.org/wiki/ROT13。——译者注

www.TopSage.com

大家网 TopSage.com

正则表达式可读性不高，原因在于它自身。普通的编程语言，通常只是将处理问题的思路直接翻译为代码而已。比如从1数到10，只需照着本意写一个 foreach 循环：

```
foreach my $i ( 1 .. 10 ) {
  print "$i\n";
}
```

但正则表达式没有这样直截了当、一一对应的指令系统，解决问题时需要动一点脑筋，而不是直接依照思路顺序翻译为代码。要处理“搜寻单引号内字串”这样的问题，写出的正则表达式确实是“天书”：

```
/'(?:\\'|.)*?'/
```

我们可以让正则表达式变得更加易读，尤其是准备将代码与他人分享，或者将来还要自己动手维护这些代码，提升正则表达式可读性的意义就更加不言而喻了。而使用尽可能简单的正则表达式，只是提高可读性的第一步。此外，Perl还提供了许多特性，允许将复杂的正则表达式拆分开来帮助理解。

1. 使用辅助空白字符和注释

一般情况下，正则表达式中的空白字符都是有意义，多一个少一个，效果截然不同。比如下例，正则表达式的普通空格用于匹配字串中的空格，充当分隔符：

```
my ( $a, $b, $c ) = /^(\w+) (\w+) (\w+)/;
```

出现在正则表达式中的换行符，也是要匹配原来字串中的换行符的：

```
$_ = "Testing
one
two"; # $_包含换行符
s/
/<lf>/g; # 将换行符替换为<lf>

print "$_\n"; # 输出 Testing<lf>one<lf>two
```

/x 选项可以同时用在匹配区和替换区，此时正则表达式解析器会忽略空白字符（被反斜线转义的空白字符除外），同时忽略注释。你可以利用这种特性，按照正则表达式的内在逻辑将其展开，分隔为不同的部分。对于上述搜索单引号内字串（包括被转义的单引号）的例子，使用 /x 选项后，添加适当的空格，便能清晰理解每部分的含义了：

```
my ($str) = /( ' (?: \\' | . )*? ' )/x;
```

而对于含有复杂的可选项以及多层分组括号的正则表达式，这种特性更加有用。使用/x，便能添上换行符并逐行解释其功用，以展开显示整个正则表达式的结构。我们甚至还可以在注释中枚举希望能匹配的文本实例：

```
my ($str) = m/
        (               # $1 的开始
        "               # 双引号字串的开始

        (?:
            \\\W                    | # 特殊字符，比如\+
            \\x[0-9a-fA-F]{2}       | # 十六进制，比如\xDE
```

www.TopSage.com

```
            \\[0-3][0-7]{2}      | # 八进制，比如\0377
            [^"\\]                 # 普通字符
        )*

        "
    )               # $1 结束

    /x;
```

2. 将复杂的正则表达式化整为零

正则表达式的优先级次于双引号内插变量（见条款 28）。我们可以利用这一点，用变量构造正则表达式。某些情况下，这会让正则表达式更为易读。结合 qr//（见条款 40），可以将子模式（较短小的正则表达式）组合成复杂庞大的正则表达式：

```
my $num = qr/[0-9]+/;
my $word = qr/[a-zA-Z_]+/;
my $space = qr/[ ]+/;

$_ = "Testing 1 2 3";
my @split = /($num | $word | $space)/gxo;
print join( ":", @split ), "\n"; # 输出为 Testing: :1: :2: :3
```

实际上这个正则表达式就是 /([0-9]+ | [a-zA-Z_]+ | [])/gxo。其中用到了只编译一次的 /o 选项（见条款 39），这样 Perl 就只会对这个复杂的组合正则表达式解析一次。

现在重写先前提到的双引号字串的例子，这次用子模式分别处理每一部分，最后合并到变量 $whole 中：

```
my $whole = do {

    # 被转义的特殊字符，例如：\", \$
    my $spec_ch = qr/\\ \W/x;

    # 十六进制字符：\xab
    my $hex_ch = qr/\\x [0-9a-fA-F]{2} /x;

    # 八进制字符：\123
    my $oct_ch = qr/\\ [0-3][0-7]{2} /x;

    # 普通字符
    my $char = qr/[^\"\\]/;

    qr/
        (
            "
            (?: $spec_ch | $hex_ch | $oct_ch | $char) *
            "
        )
        /xo;
};
#……这是最终得到的实际的正则表达式……
my ($str) = /$whole/o;
```

www.TopSage.com

如果想知道上面构建的正则表达式究竟是什么样子，可以打印变量 $whole 查看结果。正则表达式对象会将模式以字串序列的方式输出：

```
print "The regex is----\n$whole\n----\n";
```

以上利用变量构造复杂正则表达式的例子都比较简单，但应该能说明问题。而更为复杂的实际案例，还请参阅 *Mastering Regular Expressions*[①]——其中有个匹配 RFC 2822 电子邮件地址的正则表达式。

3. 要点

- 用 /x 选项添加无实际意义的空白字符及注释，提高正则表达式可读性。
- 构建正则表达式应突出显示逻辑结构。
- 由子模式组合串成的正则表达式，更易理解和维护。

条款38　避免不必要的回溯

正则表达式中的多选结构运算通常是比较慢的，这是由正则表达式引擎的工作方式决定的。当多选结构中的一个分支失败时，引擎会在字串中“回溯”到之前的位置，尝试下一个分支。

比如匹配一个名字列表，用多选结构列出所有需要的名字：

```
while (<>) {
  print
    if /\b(george|jane|judy|elroy)\b/;
}
```

该模式首先找到一个单词边界，然后尝试匹配 george。如果失败，就退回到单词边界处，尝试匹配下一个名字 jane，如果再次失败，就会再次回溯依次尝试 judy 和 elroy。如果其中任意一个名字匹配，它就会寻找另一个单词边界。

当备选项彼此相近时，回溯问题尤为严重。为说明问题，我们构造了下面这个例子，每个备选项的前半部分都相同：

```
'aaabbbccg' =
  ~/\b(aaabbbccc|aaabbbccd|aaabbbcce|aaabbbccf)\b/;
```

匹配操作符会先尝试第一个备选项，逐个字母比对，到最后一个字母 c 时才发现匹配失败。引擎只好回溯到最开始的位置，然后测试第二个备选项。这次同样是到最后一个字母才发现匹配失败，于是再次回溯，直到尝试过所有备选项后，最终发现没有一个能匹配，这才得出结论，整个匹配失败。而在此过程中，引擎做了不少工作——到头来却都是无用功。

有一种解决办法，通过建立**检索树**（trie，取自 retrieval，与单词 tree 同音谐意），将所有备选项的相同前缀提取出来，以便把回溯次数降至最低。直接使用 Regexp::Trie 模块即可，它会帮你处理余下的细节：

```
use Regexp::Trie;
```

① Jeffrey E. F. Friedl，*Mastering Regular Expressions, Third Edition*，Sebastopol，CA: O'Reilly Media，2006。

www.TopSage.com

```
my $rt = Regexp::Trie->new;
foreach (qw/foobar foobah fooxar foozap fooza/) {
  $rt->add($_);
}

my $alternation = $rt->regexp;

print "$alternation\n";
```

输出结果是该模块创建的正则表达式，以合并同类项的方式提高扫描效率，节省回溯次数：

```
(?-xism:foo(?:ba[hr]|xar|zap?))
```

使用这个正则表达式，只需内插到匹配操作符中即可：

```
$string =~ m/$alternation/;
```

从 Perl 5.10 版开始，正则表达式引擎可以自动为备选文本建立检索树，因此一般无需特意关注这方面的细节。

1. 用字符组[abc]代替多选结构

有些情况下完全没必要使用多选结构，所以应该尽量避免。比如使用多选结构枚举字符，速度肯定会严重降低。也许你以前写过这样的多选结构匹配 Perl 变量标识符号：

```
while (<>) {
  push @var, m'(($|@|%|&)\w+)'g;
}
```

由于所需匹配的全是单个字符，因此用字符组表示可选就足够了：

```
while (<>) {
  push @var, m'([$@%&]\w+)'g;
}
```

2. 避免量词引起的不必要回溯

请看正则表达式 /\b(\w*t|\w*d)\b/，它匹配以 t 或 d 结尾的单词。每次使用该正则表达式时，引擎会先寻找单词边界，然后开始尝试匹配第一个分支，它会在每行中寻找尽可能多的可以组成单词的字符，再尝试匹配字符 t。不过这个过程注定不会匹配成功，尽管刚才已经匹配了所有的单词字符，其中包括 t。于是从最后一个单词字符的位置开始回溯一位，如果恰好就是 t 的话，只要下个位置是单词边界，就算大功告成。但如果依然匹配失败，引擎就要继续回溯，直到找出符合条件的 t 为止。若回溯到起始单词边界仍不成功，引擎就会尝试其余的分支，即在当前位置尝试匹配以 d 结尾的单词。

可以看出，回溯是个循环往复的过程。虽然正则表达式原本就是要解决复杂问题的，但像上面这样大量无谓的往复实在是浪费之至。

这种写法的显著缺点在于，对于 d 结尾的单词，正则表达式引擎必须每次先行测试一遍是否以 t 结尾，等到发现匹配失败，再退回去重新测试，显然，白绕了一个大圈子。其实，完全可以去掉可选结构，直接用字符组的形式匹配单词末尾：

```
/\b\w*[td]\b/
```

www.TopSage.com

这就迈进了一大步。现在，引擎只需要对所有单词扫描一次就已足够，而单词究竟以 t、d 还是其他字母结尾，都不会影响扫描方式。

不过，这种做法还是没有完全解决回溯问题。针对每一个单词，最多只需要对其词尾的最后一个字母进行一次回溯。只要该字母不是 t 或 d，就没有必要对这个单词穷究下去了，因为在其他位置找到 t 或 d 根本不是你所需要的。

稍稍转换一下思路，问问自己："如果要找一个以 t 或 d 结尾的单词，到底应该从何处入手呢？"其实只要关注单词结尾即可：

```
/[td]\b/
```

结果很有意思。前面两条正则表达式所能做到的，这条简短的正则表达式一样也能做到，虽然它乍看之下意义不太明晰。其实我们要实现的任务就等效于寻找左边存在 0 个或多个 \w 字符的 t 或 d，而你不必关心具体是哪种 \w 字符。换句话说，只要单词边界的左侧出现一个 t 或 d，此处就是一个以 t 或 d 结尾的单词。这也是量词*出现在正则模式左侧时的特性：既然它可以匹配 0 个字符，那么用不用它，就没有什么分别。

自然地，短小的正则表达式比前两条都要快，事实上大约能快两倍左右。同时，由于它只匹配一个字符，实际回溯的次数也就更少。

3. 永不回溯的占有优先量词

避免回溯的方法之一是，明确告诉 Perl 不要回溯。Perl 5.10 引入了"占有优先"（possessive）量词。在任意量词后面加上一个 + 号，该量词就变成占有优先量词[①]，不再回溯。占有优先量词会尽可能多地匹配字符，而且从不回溯。一旦占有优先量词之后的正则表达式不能接着往下匹配，整条正则表达式即告失败。

假设在代表 DNA 结构的字串中搜索某些有趣的内容，写一条正则表达式匹配这样的模式：任意一组 GCx 这样的字符（x 代表任意字母），后接 GCG，再加任意的两个字符，最后是一个 G。或许这是一段垃圾 DNA，又或许是一段变异外星人的 DNA，但不管怎样，在下面这段代码中它将为我们清楚展示回溯现象：

```
my $string = 'GCGGCCGCAGCUGCCGCUGCAGCCGCCGCUGCCGCCGCG';

if ( $string =~ /^(?:GC.)*GCG(..G)/ ) {
  print "Matches GCG with XXG afterward!\n";
}
```

我们可以启用 re 编译指令，打开正则表达式的调试模式观察上述匹配过程。运行下列代码，观察输出：

```
use re 'debugcolor';

my $string = 'GCGGCCGCAGCUGCCGCUGCAGCCGCCGCUGCCGCCGCG';

if ( $string =~ /^(?:GC.)*GCG(..G)/ ) {
  print "Matches GCG with XXG afterward!\n";
}
```

① 占有优先量词包括：*+、++、?+、{m, n}+和{m,}+（目前仅 Java 和 PCRE 支持）。——译者注

www.TopSage.com

因为 * 是贪婪的，所以它会尽可能多地匹配。在几乎到达字串结尾时它才发现有个 GCG，但再往后就没有内容可以匹配了。于是它在贪婪的驱使下，开始回溯，从匹配位置回退三个字符尝试匹配，但仍旧失败，再回溯，再失败，周而复始，直至回到字串起始位置。虽然它在起始位置又找到了 GCG，但这一圈兜下来显然是徒劳的，因为其实在字串中匹配了最后一组 GCG 之后，它就应该能立即判断出匹配失败。本例的测试字串较短，因此情况还不算太糟糕；实际的 DNA 序列会非常长，若用这个正则表达式对数以百万计、亿万计的字串进行匹配测试，效率就是个大问题了。

为了避免回溯，我们可以用占有优先量词+。任何量词都具有对应的占有优先模式，就是在量词后添加+，以明确提示不要回溯。下面的正则表达式能在匹配失败时立即退出，而不会反复回溯：

```
if ( $string =~ /^(?:GC.)*+GCG(..G)/ ) {
  print "Matches GCG with XXG afterward!\n";
}
```

补充说明一下，这个例子也可以用排除型字符组，以免在第一组第三个字符的位置匹配 G：

```
/^(?:GC[^G])*GCG(..G)/
```

4. 要点

- 将多选结构转换为检索树，可以加快匹配速度。
- 当需要匹配一组单个字符时，最好用字符组。
- 用量词的占有优先模式避免正则表达式回溯。

条款 39　仅编译正则表达式一次

大部分匹配和替换操作的正则表达式可一次编译多次使用，因而只需在程序编译时一次性编译为 Perl 内部代码，之后就不再编译。在下面这个例子中，因为其模式固定，Perl 只对正则表达式 /\bmagic_word\b/ 编译一次，在内部运行时，每次都会使用已经编译过的代码：

```
# 计算@big_long_list 中 magic_word 的出现次数
foreach (@big_long_list) {
  $count += /\bmagic_word\b/;
}
```

如果正则表达式中使用了内插变量，比如 /\b$magic\b/ ，那么旧版本的 perl 在执行匹配操作时，每次都会重新编译，因为它无从判断变量是否变动过（而新版本改进了这点）：

```
print "Give me the magic word: ";
chomp( my $magic = <STDIN> );

# 计算@big_long_list 中 magic_word 的出现次数
foreach (@big_long_list) {
    $count += /\b$magic\b/;
}
```

每次匹配都重新编译该模式，显然浪费时间。如果确信 $magic 的内容不会改变，则可以用 /o 标志提示 Perl 只编译一次：

www.TopSage.com

```
print "Give me the magic word: ";
chomp( my $magic = <STDIN> );

# 计算@big_long_list 中 magic_word 的出现次数
foreach (@big_long_list) {
  $count += /\b$magic\b/o; # 仅编译一次
}
```

现在 Perl 对该正则表达式只编译一次。如果此时 $magic 有所改变，匹配操作仍会使用之前已经编译过的模式。

/o 标志也可用在替换操作中，用一次编译的相同模式替换列表中的多条内容：

```
print "Give me the magic word: ";
chomp( my $magic = <STDIN> );

foreach (@big_long_list) {
  s/\b$magic\b/random_word()/eo;
}
```

/o 标志还可以用在引用式的正则表达式中（见条款 40），以便预编译正则表达式。

要点

- 在正则表达式中使用内插变量要特别留意。
- 可在正则表达式中用 /o 标志表示只需编译一次。

条款 40　预编译正则表达式

马上回答——下面的 Perl 正则表达式包含哪些部分？

```
/abc/
```

答案是两部分：一个是匹配操作符，一个是内部的正则模式。匹配操作符的目的是要将正则模式同匹配目标联系起来，但匹配操作符和其内部的正则模式并不一定要同时出现。

我们可以抛开匹配或替换操作符，直接用 qr// 定义模式，然后使用这个已经编译过的正则表达式。它在语法上和定义字串没什么区别：

```
my $name = 'Buster';
my $regex = qr/\b$name\b/;
```

除了匹配操作符中内插的字串以外，其他出现在正则表达式中的元字符具有特殊含义。变量 $regex 的值变成了刚刚编译过的这个正则表达式的虚拟引用。实际上它只不过是带一点点魔法特性的普通字串而已，用 ref 函数来看就像是引用[①]，但如果真的对它解引用的话，得到的却是 undef。不管怎样，打印这个变量，就会看到下面这条正则表达式结构，一如之前我们定义的那样：

```
(?-xism:\bBuster\b)
```

① ref 返回的类型是 Regex。——译者注

www.TopSage.com

定义模式的时候同样可以加上各种修饰，比如/i、/m 或是/s。在下面这个例子中，不区分大小写，点号可以匹配换行符，因此两个名字之间可以跨行：

```
my $regex = qr/Buster(.*)Mimi/si;
```

我们可以把编译好的 $regex 用在匹配或替换操作中，就好比直接使用普通文本一样：

```
# 计算@big_long_list 中 magic_word 的出现次数
foreach (@big_long_list) {
    $count += /$regex/; # 之前就已编译过
}
```

甚至略掉匹配操作符也行，虽然看上去有点怪怪的[①]：

```
$string =~ $regex;
```

我们还可以在使用某条正则表达式之前，先用 qr// 测试一下，看能否通过编译，而不必等到应用时才发现模式本身有问题。可借助 eval 检测正则表达式编译结果：

```
my $name = '(';
my $regex = eval { qr/\b$name\b/ }
  or die "Regex failed: $@";
```

如果 $name 中的左括号（没有配对，内插后得到的模式就不完整，因此 eval 抛出错误提示。而 Perl 会很尽责地向你汇报困惑之处：

```
Unmatched ( in regex; marked by <-- HERE in m/( <-- HERE /
```

要点

- 用 qr// 操作符预编译正则表达式。
- 将预编译过的正则表达式内插到匹配操作符中使用。
- 用 eval 检测模式非法的正则表达式。

条款 41　正则表达式的性能评测

和 Perl 中许多其他部分类似，想要检测某条正则表达式的速度是否足够快，只需再写一条等效的表达式，然后作个比对就一清二楚。

我们可以使用 Benchmark 这个模块检测非捕获型括号（见条款 32）究竟有多快。下面的代码是用来解析 Apache 常规格式的 *acess_log* 日志文件，从中提取发送请求的主机名。该日志文件格式如下：

```
www.example.com - - [01/Dec/2009:21:09:42 -0600] "POST / ↵
  HTTP/1.1" 200 30447
```

代码如下，两条正则表达式外加一条简单的赋值语句作为基准，用以比对运行速度：

```
use Benchmark qw(timethese);
my @data = <>;
```

① 实际上并不奇怪，既然$regex 取 ref 得到的是 Regex，那么 Perl 自然知道它就是一个正则表达式。——译者注

www.TopSage.com

```
timethese(
  $ARGV[0] || 100,
  {
    control => q{
        foreach (@data) {
            my ($host) = 'www.example.com'
        }
    },

    mem => q{
        foreach (@data) {
            my ($host) = m/(\w+(\.\w+)+)/;
        }
    },

    memfree => q{
        foreach (@data) {
            my ($host) = m/(\w+(?:\.\w+)+)/;
        }
    },
  }
);
```

我们在 MacBook Air 上用 Perl 5.10.0 版测试，分析 100 000 行 Apache 的 *access_log* 日志文件，结果是非捕获型括号要快 22%：

```
Benchmark: timing 100 iterations of control, mem, memfr ↵
  ee...
   control: 2 secs ( 2.05 usr + 0.02 sys = 2.07 CPU)
       mem: 25 secs (24.36 usr + 0.12 sys = 24.48 CPU)
   memfree: 20 secs (19.18 usr + 0.12 sys = 19.30 CPU)
```

你的测试结果或许会有所不同，但既然要比对，就一定要确保外部平台和配置一致，才能保证测试数据的有效性。

或许你对于只有 22% 的性能提升并不太满意，但本例所测试的两条正则表达式，基本思路是完全一致的，差别仅仅在是否捕获变量上，而实际应用的时候，可能采取完全不同的思路和形式写成新的等效正则表达式，这就可能带来很大的性能提升了。还是拿这个例子来说，我们知道主机名位于日志行首，那么改用这条正则表达式看看：

```
timethese(
  $ARGV[0] || 100,
  {
    # 其他三段同上，此处从略……
    anchor => q{
        foreach (@data) { my ($host) = m/^(\S+)/; }
    },
  }
);
```

看到了吗？新的正则表达式比之前的都要快上许多：

```
Benchmark: timing 100 iterations of anchor, control, ↵
  mem, memfree...
```

www.TopSage.com

```
 anchor: 11 secs (10.78 usr + 0.07 sys = 10.85 CPU)
control: 2 secs ( 2.02 usr + 0.01 sys = 2.03 CPU)
    mem: 25 secs (24.69 usr + 0.19 sys = 24.88 CPU)
memfree: 20 secs (19.34 usr + 0.14 sys = 19.48 CPU)
```

很多情况下，加不加定位锚符，正则表达式的模式是否足够精简，都会大大影响正则表达式的执行速度。

要点

- 可以用 Benchmark 模块比较正则表达式的效率。
- 建立一条简单朴素的基准比对语句，作为性能参考基线。
- 如果小改小修没有带来显著改善，可以试着另辟蹊径，推倒重来。

条款 42　不要滥造正则表达式

在文本中搜索数字似乎是小菜一碟，类似于 /\d+/ 这样的简单模式就可以胜任。但十进制数字中的小数点什么的该如何处理？还有每三位数字间的千位分隔符呢？再进一步，正负号呢？指数记号呢？类似的问题不费力就能找出一大堆。

如果不是为了做练习，你会不会花时间写删除空白字符或搜索电子邮件地址的正则表达式？对这类常见问题，其实早已有人写好了通用模式，做成漂亮的 Regex::Common 模块放在 CPAN 上，就等着你来用：

```
use Regexp::Common;
my $text =
  'Absolute zero is -459.67 on the Fahrenheit scale';
print "$1 is [beyond] freezing!\n"
  if $text =~ /$RE{num}{real}{-keep}/;
```

Regexp::Common 导出了一个名为 %RE 的散列表，包含各种经过预定义、随取随用的正则表达式，上面提到的 $Re{num}{real} 就是一例。而最后的键 {-keep} 其实隐含了一点魔法，Regexp::Common 会在幕后解析这类用作选项的键，而这里的 {-keep} 表示保存捕获内容。

它甚至还提供了面向对象接口供人使用，只需指明需要的键，调用 match 方法即可。面向对象的接口一般只匹配不捕获，不过这不影响你构造“文明用语”过滤器：

```
use Regexp::Common;

my $text = 'shpx';
$text =~ tr/A-Za-z/N-ZA-Mn-za-m/;

while (<>) {
  print "Kiss your mother with that mouth?\n"
    if $RE{profanity}->matches($text);
}
```

上述代码会对爆粗口的用户稍作警示。顺便提一句，如果想从 profanity 源码中翻出些市井粗口找点乐子的话，恐怕不会如愿，因为作者已经对脏话打上了“马赛克”[①]，以便评为大众

① 还是请翻看源码，实际作者只是作了简单的 ROT-13 转译，但写法上和之前介绍的略有不同。——译者注

www.TopSage.com

大家网 TopSage.com

级（G-rated），不致遭人诟病。

由于 Regexp::Common 是通过将不同功用的模式分割为若干独立的子模块实现的，所以有时候查找自己需要的正则模式并不太容易。但这么做是有道理的，你可以自由选择要载入的模式，比起一股脑儿地全部载入，这样既节约了时间，也降低了负载。

此外，独立子模快的形式还可以让你自由定制，为 Regexp::Common 模块安装额外“插件”。比如 Regexp::Common::Email::Address 模块就是一个独立的插件模块，在加载 Regexp::Common 时可以选择导入：

```
use Regexp::Common qw(Email::Address);

my $text =
  'my email address is josh+spam@example.com, okay';

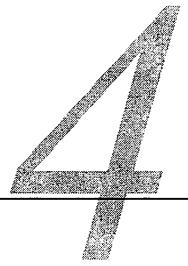
if ( $text =~ /$RE{Email}{Address}{-keep}/ ) {
  $text = $RE{ws}{crop}->subs($1);
  print "$text\n";
}
```

要点

- 善用 Regexp::Common 模块，而不是对已十分常用的正则表达式随意滥造。
- 通过载入插件模块的方式，扩充 Regexp::Common 支持的正则模式。

www.TopSage.com

第4章 子程序



作为一门动态语言，Perl的强大之处源自于它灵活的子程序。要定义一个子程序很简单，我们无需告诉它将会收到多少参数，以及每个参数的数据类型，直接把数据列表传给子程序就好了，之后再来决定具体如何处理。尽管Perl是一门非常松散的语言，它的灵活和轻巧可以让我们在最后关键时刻再作决定，但也有需要注意的地方，否则太过散漫也容易出问题。

Perl里面的子程序根本不需要提前定义。我们可以在程序运行时动态新建一个，也可以在之后重新定义。子程序本身也可以用来构建其他子程序。而每一个子程序都能包含各自独立的私有数据。定义好的子程序可以保存为引用，然后像其他数据一样作为参数传递。

善用活用Perl子程序，复杂问题也会变简单。

条款43 理解my和local之间的差异

my和local之间的差异是Perl中非常微妙和复杂的方面之一。之所以说它微妙，是因为能观察到my和local在功能上有所区别的场合十分少见。但它们的不同行为确实会带来不一样的运行结果，不弄清楚的话很容易混淆使用，最终导致程序运行偏离预期。

有时候人们谈到my和local之间的区别可能会这么说："my变量只存活于声明时所在的子程序内部，而local变量在子程序内部调用的其他子程序里，也是可以访问的。"但其实这种说法是错误的：my和子程序根本就没关系，local也一样。非但如此，它们之间的区别也和子程序没有关系。虽然看起来好像是有的，但我们马上就会看到，实际完全是另外一回事。不管怎么说，本节要详细讨论local和my，它们在子程序中都扮演着非常重要的角色。

1. 全局变量

在Perl里面，所有的变量、子程序和其他可以被命名的实体默认都拥有包作用域（或称为"全局作用域"）。也就是说，它们存在于当前包的符号表中。花括号、子程序或是文件，是不会创建local变量的。

大多数情况下，Perl会在编译阶段将全局名称放到合适的包符号表中。那些在Perl编译时没法获知的名称会在执行时插入到符号表中。让我们运行一个名为tryme的程序来验证一下：

```
print join " ", keys %::;
```

www.TopSage.com

```
$compile_time;  # 编译时创建
${"run_time"};  # 软引用，运行时创建
```

当我们运行这个程序时，我们会看见 Perl 在符号表中列出跟踪的所有东西：

```
ARGV 0 FileHandle:: @ stdin STDIN " stdout STDOUT $
stderr STDERR _<perlmain.c compile_time DynaLoader::
_<tryme ENV main:: INC DB:: _ /
```

注意，在这个例子中，标识符 compile_time 在符号表出现的位置，要比变量 $compile_time 实际在运行时出现的位置早。如果我们之前对 Perl 的这种编译型的性质不确定，这个例子就是个很好的证明。

也许在我们编程生涯（或者说爱好，或者其他什么说法）刚开始的时候，就被告知全局变量很糟糕：好程序不应该用很多全局变量，因为全局变量就像隐藏接口，指不定什么时候就变掉了，代码也会难于理解和修改，甚至会导致编译器无法优化等诸多问题。

如果你曾经写过几百行代码的程序，特别是在团队协作开发的情况下，对此多少会有一点共鸣，也就能够理解和需要 Perl 支持本地变量的特性。

当然，Perl 确实支持本地变量，事实上，它比很多其他语言支持得都要好。很多语言只提供一种单一的创建本地变量的机制，而 Perl 却提供了两种，一个是 my，一个是 local。

2. my 变量词法作用域（编译时）

Perl 的 my 操作符用于创建词法作用域[①]变量。通过 my 创建的变量，存活于声明开始的地方，直到闭合作用域结尾。闭合作用域指的可以是一对花括号中的区域，可以是一个文件，也可以是一个 eval 字符串。闭合作用域是词法作用域，仅在编译时对程序源代码扫描过程中确立。换种说法，my 变量的作用域，实际只需通过阅读源代码就能判断。下面的代码同时用到了全局变量 $a 和词法变量 $a：

```
$a = 3.1415926;  # 全局的
{
  my $a = 2.7183; # 词法的
  print $a;       # 2.7183
}
print $a;         # 3.1415926
```

这里花括号外面用到的变量 $a 是全局变量 $a，而在花括号内部的 $a 是词法变量 $a，它的作用域仅限于花括号内。

这种方式是编程语言最常见的处理作用域的方式。既然这是在 Perl，听到看到一些奇思妙想应该没什么好惊讶的。而事实上，这里存在的不仅仅只是这些奇思妙想。

回顾刚才我们对符号表的检查。下面的程序和本条款刚开始的那个很像，但输出内容的顺序有些不同，这次用的是 my：

① 词法作用域，就是说解释器在解析程序源代码时，按照规定的语法理解代码意义，将相应的变量设置为仅在某个区域内存活有效。这种设定是按照字面上的代码范围进行的，所以叫做词法作用域，它在编译阶段确立。

——译者注

www.TopSage.com

```
my $compile_time; # 词法变量
$compile_time;
print join " ", keys(%::);
```

再次运行，输出信息中已经没了 compile_time，请对照之前的输出：

```
ARGV 0 FileHandle:: @ stdin STDIN " stdout STDOUT $ stderr
STDERR _<perlmain.c DynaLoader:: _<tryme ENV main:: INC
DB:: _ /
```

让我们看看其他情况：

```
$compile_time;     # 不是 my 变量
my $compile_time; # 但这个是 my 变量
print join " ", keys(%::);
```

运行上面这段程序后，会看到 compile_time 又回来了：

```
ARGV 0 FileHandle:: @ stdin STDIN " stdout STDOUT $
stderr STDERR _<perlmain.c compile_time DynaLoader::
_<tryme ENV main:: INC DB:: _ /
```

这些例子证实了 my 变量并不“存在”于符号表中。在先使用 my $compile_time 的例子中，只有一个变量被命名为 $compile_time，它从未出现在包的符号表中。而后面的例子中，有两个独立变量被命名为 $compile_time：一是存在于符号表中的全局变量，一是不在符号表中 my $compile_time 变量。

我们总能通过限定名称访问包变量值。限定名称（包含 :: 符号）总是指向符号表中的变量，比如：

```
# 限定名称变量对比 my 变量
{
  my $a = 3.1416;
  $main::a = 2.7183;
  print "(inside) a = $a\n";
  print "(inside) main::a = $main::a\n";
  print "(inside) ::a = $::a\n";
}
print "(outside) a = $a\n";
```

输出显示 Perl 在每一步是怎么看待$a 的：

```
(inside) a = 3.1416
(inside) main::a = 2.7183
(inside) ::a = 2.7183
(outside) a = 2.7183
```

符号表还被用来做很多其他事情，包括软引用和 typeglob。由于 my 变量不在符号表中，我们没法用上面两种中的任一种方式访问。下面是一个关于软引用的演示，就算同名词法变量在同一作用域内也没关系，因为在软引用中总是用符号表维护变量：

```
# 软引用变量对比 my 变量
my $a = 3.1416;
${'a'} = 2.7183;
print "my a = $a\n";
print "{a} = ${'a'}\n";
```

www.TopSage.com

从运行结果可以看出，Perl在每个print语句用到的是不同版本的$a：

```
my a = 3.1416
{a} = 2.7183
```

typeglob的工作方式也是如此。实际上，正如下面这个例子所演示的一样，无论是typeglob、软引用，还是限定变量名，都不会指向词法（my）变量：

```
# typeglob对比my变量
$a = 2.7183;
my $a = 3.1416;
*alias = *a;
print "my a = $a\n";
print "alias = $alias\n";
print "{a} = ${'a'}\n";
print "::a = $::a\n";
```

输出显示了Perl在每一步是怎么看待$a的：

```
my a = 3.1416
alias = 2.7183
{a} = 2.7183
::a = 2.7183
```

注意，*alias这个typeglob指向全局的*a，尽管typeglob的赋值在my $a之后。实际上，这和赋值操作的先后没有关系，在哪都一样——typeglob永远指向符号表中的东西，而my变量则不会出现在其中。

关于其他“技巧”，得等我们了解local变量后才能继续讨论。

3. local的运行时作用域

Perl的另一种作用域机制是local，它比my的历史更悠久。事实上，my是直到Perl 5时才引入的。也许你会好奇，local到底哪里不好，以至于Larry Wall认为值得加入一个完全不同的作用域机制作为补充？

要回答这个问题，得先看看local是如何工作的。在某些情境下，my的好处会自然显露出来。

local是运行时作用域机制。和my不同，my基本上是编译时在私有符号表中创建新变量，而local则是在运行时起作用：它会将参数的值保存在一个运行时栈中，当执行线程离开所在作用域后，原先作用域外暂存的变量会被恢复。

乍看下，local和my做的事情貌似非常相似。下面是与之前类似的例子，但将my替换成了local：

```
# 基本的local用法
$a = 3.1416;
{
  local $a = 2.7183;
  print $a; # 2.7183;
}
print $a;   # 3.1416;
```

尽管这个看起来和用my的例子差不多，输出结果也一样，但在Perl内部，事情却是完全不同的。

www.TopSage.com

在 my 的例子中，像我们所见到的那样，Perl 创建了在运行时无法使用名字访问的独立变量。换句话来说，它不存在于包符号表中。在执行内部的代码块时，外面全局的那个 $a 仍然存在于符号表中，并且值仍为 3.1416。

而在 local 的例子中，尽管 Perl 将 $a 的当前内容保存在运行时栈中，Perl 还是将 $a 替换成了新的内容。当程序离开闭合的代码块后，Perl 将 local 所保存的值恢复。在整个例子中，只有一个名为 $a 的变量。

为了更好地进行描述，让我们用软引用来探视一下符号表：

```
$a = $b = 3.1416;
{
  local $a = 2.7183;
  my $b = 2.7183;
  print "IN: local a = $a, my b = $b\n";
  print "IN: {a} = ${'a'}, {b} = ${'b'}\n";
}
print "OUT: local a = $a, my b = $b\n";
```

运行后输出：

```
IN: local a = 2.7183, my b = 2.7183
IN: {a} = 2.7183, {b} = 3.1416
OUT: local a = 3.1416, my b = 3.1416
```

瞧，多有趣。使用软引用来查看全局 $a 的技巧，对 my 有用而对 local 却毫无用处。其实理应如此：my 创建了不同的变量，而 local 只是将已存在的变量值暂时保存起来罢了。

由于 local 是运行时而不是编译时，因此 local 对全局变量所做的改动，在其词法作用域之外仍能观察到。最为人熟知的典型应用就是嵌套子程序调用：

```
$a = 3.1416;
sub print_a { print "a = $a\n" }

sub localize_a {
  print "entering localize_a\n";
  local $a = 2.7183;
  print_a();
  print "leaving localize_a\n";
}
print_a();
localize_a();
print_a();
```

运行后输出：

```
a = 3.1416
entering localize_a
a = 2.7183
leaving localize_a
a = 3.1416
```

这样的例子常用来演示 local 的用处，看起来就好像 local 和子程序调用有着紧密关系，但其实就如我们之前所说的，二者毫无瓜葛。

4. 何时该用 my

通常情况下，我们应该用 my 而不是 local。理由之一是 my 比 local 快，因为 local 将值存入栈的操作，是需要消耗时间的：

```
use Benchmark;
timethese(
  1_000_000,
  {
    'local' => q{ local $a = $_; $a *= 2; },
    'my'    => q{    my $a = $_; $a *= 2; },
  }
);
```

输出显示，my 同样比较快：

```
Benchmark: timing 1000000 iterations of local, my...
     local: ( 0.98 usr +  0.01 sys = 0.99 CPU) ↵
         @ 1010101.01/s
        my: ( 0.55 usr + -0.00 sys = 0.55 CPU) ↵
 @ 1818181.82/s
```

而且 my 也比较容易理解，它也不会像 local 那样存在奇怪的“非局部化”副作用。

另一个使用 my 的理由是，它所创建的词法变量是 Perl 闭包的实现基础（见条款 49）。

5. 何时该用 local

而用 local 的理由之一是，还有很多古老的 Perl 4 风格的代码在使用它。将 local 换成 my 并不只是用文本编辑器搜索并替换那样简单——我们需要对每个用到 local 的地方进行检查，看它是否有特别用到 local 的哪个“特性”。其实只要程序运行没问题，留着 local 又有何妨呢。

当然，也有一些必须用 local 才能解决的问题。

大部分 $ 开头的变量，或者其他 Perl 特别对待的变量，只能用 local 来进行本地化（尽管 Perl 5.10 引入了词法变量的 $_ 。见条款 15）。而用 my 试图对特殊变量进行本地化是错误的：

```
my $contents = do { my $/; open ... }; # 错误!
```

在其他一些不能用 my 的地方我们可以用 local，比如修改其他包中的变量：

```
package foo;
$a = 3.1416;
{
  package main;
  local $foo::a = 2.7183;

  package foo;
  print "foo::a = $a\n";
}

package foo;
print "foo::a = $a\n";

% tryme foo::a=2.7183 foo::a=3.1416
```

我们也可以对数组及散列的元素使用 local。看起来是有点奇怪，但确实如此。甚至对切片，

www.TopSage.com

我们也可以用 local 本地化：

```
@a = qw(Jolly Green Giant);
{
  local ( @a[ 0, 1 ] ) = qw(Grumbly Purple);
  print "@a\n";
}
print "@a\n";
```

上面的程序输出：

```
Grumbly Purple Giant
Jolly Green Giant
```

我们也可以对 typeglob（见条款 118）使用 local。理论上，基本任何值都能用 local 本地化，但实际运用时会受限于当前 Perl 版本。比如我们不能对像 $$a 这样解引用的值使用 local，至少在写本书时是如此。

6. local 和 my 对列表的操作

local 和 my 的语法相同，无论是单个标量，还是数组或者散列，都可以用这两种类型声明：

```
# 一些使用 local 和 my 的例子
local $scalar;
my @array;
local %hash;
```

我们可以在本地化某个变量值的时候对它初始化：

```
local $scalar = 3.1416;
my @array = qw(Mary had a little lamb);
local %hash = ( H => 1, He => 2, Li => 3 );
```

如果把 my 和 local 的参数用圆括号括起来，参数就成了列表，Perl 会在列表上下文中对赋值操作进行求值：

```
local ( $foo, $bar, $bletch ) = @a; # @a 中的前三个元素
```

当心常见的列表赋值“陷阱”：

```
# 错误——别忘了圆括号!
# $bletch 得到的是 @a 的大小；只有 $foo 被本地化了!
local $foo, $bar, $bletch = @a;

# 错误——本地化了 @a、@b，但只有 @a 被赋值
my ( @a, @b ) = @c;

# 错误——读入的是所有输入内容
my ($a) = <STDIN>;
```

7. 要点

- 用 my 创建作用域内私有词法变量。
- 用 local 可以临时修改全局变量的值。
- 对于大部分特殊变量，得用 local 本地化。

www.TopSage.com

条款44　若非必要请勿直接使用@_

和许多其他编程语言不同，Perl没有内建对命名参数，也没有那种所谓“正式”参数的支持。子程序的参数都是通过数组变量@_传递的。对参数的名称及其结构的有效性、参数数据的合法性，都应该由子程序作者负责分析检验。

一般来讲，我们都是在子程序开始时复制传进来的参数，并分别命名，常用my声明参数变量：

```
sub digits_gone {
  my ($str) = @_;
  $str =~ tr/0-9//d;   # 删除字符串中的数字
  $str;                # 返回修改后的字符串
}
```

读取子程序传入参数的惯用方式是用shift函数逐个提取，或使用列表赋值一次性获取：

```
sub char_count {
  my $str   = shift;
  my @chars = @_;
  my @counts;
  for (@chars) {
    push @counts, eval "\$str =~ tr/$_//";
  }
  @counts; # 返回计数列表
}
```

@_的元素实际上就是我们传进来的参数的别名，所以修改@_的元素其实也就是修改了子程序外部参数变量的值，我们称之为“引用式调用”语法。正是因为如此，如果外部参数实际为只读的话，在子程序内部修改参数会导致错误：

```
sub txt_file_size {
  $_[0] .= '.txt' unless /\.txt$/;
  -s $_[0];
}
```

如果调用这个子程序时传入普通字串，比如txt_file_size "test"，它就会报错，提示试图修改只读值text。

但有时候这种别名式的“特性”也确实挺有用，比如借助子程序修改原始参数，让它负责清理数据：

```
sub normalize_in_place {
  my $max = 0;
  for (@_) { $max = abs($_) if abs($_) > $max }
  return unless $max;
  # 注意$_是“别名的别名”，这也能工作
  for (@_) { $_ /= $max }
  return; # 返回空
}

my ( $x, $y, $z ) = 1 .. 3;
```

www.TopSage.com

```
normalize_in_place $x, $y, $z;
printf( ( "%.2g " x 3 ) . "\n", $x, $y, $z );
```

输出结果显示，我们对 normalize_in_place 的调用会影响 $x、$y 和 $z 的值：

```
0.33 0.67 1
```

如果要优化性能，直接使用 @_ 确实会比复制一份更快，无论如何，变量的复制总是会占用很多时间。如果一定要这么做，记住对数组取下标的操作会比较慢，我们可以用类似 foreach、grep 和 map 这些结构来遍历数组，这样就不用重复通过下标取值。究竟选择哪种实现方式，最好还是写两个或多个不同版本的代码，进行基准测试，一一比对后再作决定。

虽然子程序的参数是以别名方式进行传递的，但数组作为参数传进来后，会被展开为列表，所以就算修改收到的参数元素，也不会影响原来的数组元素：

```
sub no_bad {
  for $i ( 0 .. $#_ ) {
    if ( $_[$i] =~ /^bad$/ ) {
      splice @_, $i, 1;
      print "in no_bad: @_\n";
      return;
    }
  }
  return;
}

my @a = qw(ok better fine great bad good);
no_bad @a;
print "after no_bad: @a\n";
```

输出显示，虽然我们在 no_bad 里修改了@_，但子程序返回后，@a 的值还是和之前一样，没有变化：

```
in no_bad: ok better fine great good
after no_bad: ok better fine great bad good
```

最后还要提一点，如果未加参数调用子程序，那么子程序会有一个默认的空的 @_ 数组。而如果以 & 符号调用子程序并不加圆括号时，情况又会不同，它会继承当前环境中的 @_ 数组，请看：

```
sub inner {
  print "\@_ = @_\n";
}

sub outer {
  &inner; # 这是唯一能正常工作的语法
}

outer 1 .. 3; # 打印出@_ = 1 2 3
```

要点

- 请避免修改 @_ 的值，因为它会改变原始数据。
- 如果不需要对原始参数进行修订，那就没必要复制 @_ 中的值。

www.TopSage.com

大家网 TopSage.com

条款45　用 wantarray 按需返回列表

我们知道，子程序是可以返回标量值或列表值的。有时候需要看情况分别返回这两种类型数据，而对于标量上下文和列表上下文的概念和用法，我们在之前也有过介绍（见条款 12）。可有时候脑海一片空白，对下面这段代码的工作原理就会感到迷惑：

```
sub sorted_text_files {
  my $dir = shift;
  opendir my ($dh), $dir or die "eh?: $!";

  # 此处代码和 wantarray 无关，但只有
  # 加上目录前缀才能测试是否为文本文件
  my @files = grep { -T } map { "$dir/$_" } readdir $dh;
  sort @files;
}
```

在 print 语句中调用 sorted_text_files，可以得到一个排过序的文件列表：

```
print join ' ', sorted_text_files '/etc';
```

由于输出结果在我们意料之中，所以大概不会觉得这里有什么奇妙的地方。然而如果在标量环境中使用 sorted_text_files，结果就不同了：

```
print join ' ', scalar( sorted_text_files '/etc' );
```

这种写法，不会输出任何内容。

但如果将最后一行中的 sort @files 改成 @files = sort @files，或许看起来会更明显一点。这样最后会得到一个数字，即文件数量，而不是像之前一样什么都没有。

这里我们所看到的，就是 Perl 对子程序返回值进行求值的结果。子程序的返回值处于列表还是标量上下文，取决于我们调用该子程序时所在的上下文环境。Perl 会将求值上下文环境作用于被当作是返回值的任何表达式上。

假如需要知道调用时的上下文环境，我们可以用 wantarray 操作符（wantarray 这个名字不怎么好，因为字面上好像暗指必须是数组），如果子程序调用时处于列表上下文，它会返回真。所以接下来修改 sorted_text_files 子程序，让它在标量上下文返回连接成字符串的文件名列表，可以这样实现：

```
sub sorted_text_files {
  my $dir = shift;
  opendir my ($dh), $dir or die "eh?: $!";

  # 此处代码和 wantarray 无关，但只有
  # 加上目录前缀才能测试是否为文本文件
  my @files = grep { -T } map { "$dir/$_" } readdir $dh;
  if (wantarray) {     # 判断是列表上下文还是标量上下文
    sort @files;       # 列表上下文时，返回列表
  }
  else {
    join ' ', sort @files; # 标量上下文时，返回字串
  }
```

www.TopSage.com

```
}
```

wantarray 偶尔也会用来做选择性判断。如果我们想搞清楚 grep 区块的参数是在标量还是列表上下文中求值，可以用类似下面的方式：

```
sub how { print wantarray ? "arrayish" : "scalarish" }
grep { how() } 1;
```

它的输出就像是帮我们回答了这个问题：

```
scalarish
```

1. 空上下文

还有第三种上下文环境，即空上下文环境。对于这种情况，我们可以让子程序不返回任何类型的值：

```
sorted_text_files('/etc');
```

显然这是空上下文，没有任何目标会用到它的返回结果，所以改成下面这种写法之后，sorted_text_files 直接退出，不做任何处理，因为做了也是白做。空上下文时，wantarray 会跳过后续代码，直接返回 undef：

```
sub sorted_text_files {
  return unless defined wantarray;
  # 实际处理代码同上，此处从略
}
```

2. 精细控制

想要知道从子程序返回的数据在运行中究竟发生了什么，可以用 Contextual::Return 模块探查。拜 Damian Conway[①]赋予的魔力所赐，我们可以更详细地知道数据求值环境。以下摘录了模块文档中的部分代码：

```
use Contextual::Return;

sub handle_everything {
  return SCALAR { 'thirty-twelve' }
  BOOL          { 1 }
  NUM           { 7 * 6 }
  STR           { 'forty-two' }
  LIST          { 1, 2, 3 }

  HASHREF       { { name => 'foo', value => 99 } }
  ARRAYREF      { [ 3, 2, 1 ] }

  GLOBREF       { \*STDOUT }
  CODEREF {
    die "Don't use this result as code!";
  };
}
```

① Damian Conway 是该模块作者。——译者注

www.TopSage.com

虽然 Contextual::Return 很酷，但也不要滥用。

3. 要点

- 让子程序的返回值对上下文作出正确反应。
- wantarray 函数会告诉我们子程序所处的上下文环境。
- 如果对返回值的掌控有更高要求，可以试试 Contextual::Return 模块。

条款 46　传递引用而非副本

“老式”的子程序参数传递方式有两个缺点：首先，尽管我们可以修改参数中的元素，但却无法修改数组或散列参数变量本身；其次，将数组和散列复制到 @_ 花费时间过长。而通过传递变量引用（见条款 58）的方式，我们可以将这些缺点都克服掉。

1. 传递引用参数

当我们给子程序传递参数时，Perl 会将它们别名化后放入 @_ 。这么做是出于性能方面的考虑。之后如果我们从 @_ 提取出来保存到变量中时，Perl 才会真的复制它们。所以传递的参数越多，Perl 要做的工作也就越多。比如下面的子程序，求一个列表中数字总和，从参数中直接获取列表各个元素存入数组：

```
sub sum {
  my @numbers = @_; # 进行复制操作
  my $sum = 0;
  foreach my $num (@numbers) { $sum += $num }
  $sum;
}
```

如果调用这个子程序的人一次给出很多很多数字呢？

```
sum( 1 .. 100_000 );
```

Perl 不得不在 sum 中复制 100 000 个元素到数组变量 @numbers 中去，尽管不用修改元素的值，但依然制造一份副本无疑是一种浪费。如果只传递一个指向数组的引用，那就可以省掉这堆无谓工作。只需小小的改动就能修正：

```
sub sum {
  my ($numbers_ref)    = @_; # 进行复制操作
  my $sum              = 0;
  foreach my $num (@$numbers_ref) { $sum += $num }
  $sum;
}
```

由于 Perl 的参数永远是展开的列表，所以子程序对原始数据结构可以说一无所知。如果参数列表由两个或多个数组构成，那么子程序最后只会看到两个数组展开后串接合并到一起的完整列表。为了有所区分，我们可以分别传递它们的引用：

```
process_arrays( \@arrayA, \@arrayB );
```

而在子程序中，我们会得到一个包含两个数组引用的列表，随后便可逐个进行处理：

www.TopSage.com

```
sub process_arrays {
  my (@array_refs) = @_; # 所有数组的引用

  foreach my $ref (@array_refs) {
    ... 数组的处理 ...;
  }
}
```

任何一种数据结构的引用，都可以用这样的方式传递。而子程序内部，只需要逐个提取以正确方式使用即可：

```
process_refs( \@array, \%hash, \&sub_name );
```

当然，不仅仅是数组和散列，如果有一个很长的字符串，同样也可以通过传递标量引用的方式避免在子程序内复制字符串。请看下面的例子，通过引用，我们在子程序内部直接操作外部字串，无需复制：

```
process_big_string( \$string );

sub process_big_string {
  my $string_ref = shift;

  $string_ref =~ s/\bPERL\b/Perl/g;
}
```

2. 返回引用参数

返回结果和传入参数的过程恰好相反，既然传递引用能免去传入参数时的复制操作，那么返回数据时同样也可以采取传递引用的方式返回，而调用子程序的地方，最终一样也会得到一个展开的列表。特别是要返回的数据结构复杂庞大时，更应该直接返回引用。比如下面这个子程序，读取文件后把所有内容保存在标量变量内，最后仅仅返回该标量的引用：

```
my $string_ref = slurp_file($file);
print "The file was:\n$$string_ref\n";

sub slurp_file {
  my $file = shift;

  open my ($fh), '<', $file or die;
  local $/;
  my $string = <$fh>;

  \$string;
}
```

当然我们也可以在子程序中返回多个数据，这就和给子程序传递参数一样，以便区分：

```
my ( $array_ref, $hash_ref ) = make_data_structure();

sub make_data_structure {
  # ...
  return \@array, \%hash;
}
```

www.TopSage.com

3. 为了速度，传递 typeglob

在引用出现之前，程序员们会在需要传递数组和散列时，借助于 typeglob 的方式。下面就是一个这样的例子，利用 typeglob 获取两个数组的引用：

```
sub two_arrays {
  local *a1 = shift; # 创建私有的 a1 和 a2
  local *a2 = shift;

  # 现在，可以对@a1 和@a2 做任何形式的处理……
}

our @a = 1 .. 3;
our @b = 4 .. 6;
two_arrays *a, *b;
```

而现今，我们不需要再写这样的代码，但在处理之前遗留下来的代码时，或许会碰到类似的写法。

4. 用 local * 对引用参数本地化

原本出于速度考虑而接受引用参数的子程序，有时候也会因此变得更慢，因为重复不断地对这些参数解引用也是需要花时间的。而某些人可能觉得解引用的语法比较难懂，甚至容易在这个问题上犯迷糊。

下面的子程序接受两个数组，依次比对每个位置大小，取大值后构成新的列表返回：

```
sub max_v {
  my ( $a, $b ) = @_;
  my $n = @$a > @$b ? @$a : @$b; # 最长数组的长度
  my @result;
  for ( my $i = 0 ; $i < $n ; $i++ ) {
    push @result, $$a[$i] > $$b[$i] ? $$a[$i] : $$b[$i];
  }
  @result;
}
```

这么多双重美元符很难看，对吧?

要解决这个问题，方法之一就是对数组取别名。取别名可以用赋值给 typeglob 的方式，这对任何数据类型的引用都适用，随后便可以使用别名变量：

```
sub max_v_local {
  local ( *a, *b ) = @_;
  my $n = @a > @b ? @a : @b;
  my @result;
  for ( my $i = 0 ; $i < $n ; $i++ ) {
    push @result, $a[$i] > $b[$i] ? $a[$i] : $b[$i];
  }
  @result;
}
```

只要理解了第一条赋值语句，整个子程序的工作原理就很好理解了。而且它比之前第一个版本也更快一些。至少当我测试这个例子时，可以观察到大约有 10% 的速度提升——改进虽然不

www.TopSage.com

TopSage.com

是特别大，但也算显著了。

5. 要点

- 当传递大数据结构或字符串时，可以用参数引用避免复制操作。
- 通过给子程序传递引用来操作原始数据。
- 当需要完整传递数组或散列时，可以用传递引用的方式。

条款 47　用散列传递命名参数

尽管 Perl 没有提供自动命名参数的传递方法（或者换句话说，没有提供所谓的“正式参数”的概念，见条款 48），但我们在调用子程序时仍然有很多方法，可以同时传递包含名字和值的参数列表。

这些方法需要我们在函数内部做一些额外的处理参数列表的工作。换言之，这不是 Perl 内置特性，但它依然有它的妙处。不同的命名参数实现适用于不同场合，而无论是哪一种，在 Perl 里面写起来都很简单。

一个简单的方法就是根据参数列表构建一个散列：

```
sub uses_named_params {
  my %param = (
    foo => 'val1',
    bar => 'val2',
  );
  my %input = @_; # 以散列的方式获得参数

  # 将读入的参数和默认参数结合起来
  @param{ keys %input } = values %input;

  # 现在可以使用诸如 $param{foo}、$param{bar}等形式的参数了
  ...
}
```

现在我们可以像定义散列那样，以键值对的形式调用子程序 uses_named_params 了：

```
uses_named_params( bar => 'myval1', bletch => 'myval2' );
```

代码并不多，并且也很简单。这个使用散列作为参数的用法非常自然。

也许你想让别人调用子程序时，可以随意选用占位参数或命名式参数。最简单的实现方式就是给参数名加上减号前缀。然后在子程序内部先检查第一个参数是否以减号开头，如果是，则将全部参数作为命名参数处理。这有一种简单直接的实现方式：

```
sub uses_minus_params {
  my %param = ( -foo => 'val1', -bar => 'val2' );
  my %input;

  if ( substr( $_[0], 0, 1 ) eq '-' ) {
    # 将命名参数读入散列
    %input = @_;
```

```
  }
  else {
    my @name = qw(-foo -bar);
    # 给占位参数命名，并存入散列
    %input = map { $name[$_], $_[$_] } 0 .. $#_;
  }
  # 覆盖默认参数
  @param{ keys %input } = values %input;

  # 现在可以使用诸如 $param{-foo}、$param{-bar} 等形式的参数了
}
```

现在占位参数和命名参数都可以调用该子程序，效果也一样（但最好选定并总是使用同一种方式）：

```
uses_minus_params( -foo => 'myval1', -xtra => 'myval2' );
uses_minus_params( 'myval1', 'myval2' );
```

但请注意，避免使用单个字符作为参数名——比如说，-e 和 -x。这样不但精简过了头，还容易和文件测试操作符（见条款 51）混淆。

像这种处理方式，即使子程序中表示参数的散列以减号为前缀表示键名（比如 $param{-foo}、$param{-bar}），乍看起来可能比较好笑，“这真的是 Perl 程序吗？”实际上 Perl 会将前缀为减号的裸字全部当作字符串，只不过第一个字符是减号罢了。

一般来说这么做很方便，但也有缺点。首先，虽然 Perl 会特殊对待以减号开头的标识符，但仅仅当它出现在 => 符号左边或单独写在花括号内时是这样处理的。所以，像 -print 这样的参数，我们就不得不用引号引起来，免得最后变成 -1[①]（在此之前还会打印当前 $_ 的值）。其次，如果我们要用占位参数，并且要传递的第一个参数恰好是负数，那就必须在该字符串前补上空白字符，以绕开命名参数的判断，不用说，这么做实在是够难看的。

虽然有些地方不会出现这样的问题，但却有更多场合会遇到。倒也不是没有办法，可以考虑将命名参数以匿名散列表的方式传递。

```
sub uses_anon_hash_params {
  my %param = ( foo => 'val1', bar => 'val2' );
  my %input;

  if ( ref $_[0] eq 'HASH' ) {
    # 将命名参数读入散列
    %input = %{ shift() };
  }
  else {
    my @name = qw(foo bar);
    # 给占位参数命名并存入散列
    %input = map { $name[$_], $_[$_] } 0 .. $#_;
  }

  # 覆盖默认参数
```

① 先是执行 print $_ 打印当前默认参数的值，然后 print 函数返回成功操作状态 1，连接之前的减号，构成数字 -1 返回。——译者注

www.TopSage.com

```
  @param{ keys %input } = values %input;

  # 现在可以使用诸如 $param{foo}、$param{bar} 等形式了，比如：
  for ( keys %param ) {
    print "$_: $param{$_}\n";
  }
}
```

现在，使用命名参数和占位参数的语法，看起来就像：

```
uses_anon_hash_params( { foo => 3, test => 10 } );
uses_anon_hash_params( -123, 345 );

# 抑或是...
uses_anon_hash_params { foo => 3, test => 10 };
```

上面这个子程序开头部分的写法还是稍显复杂，直接拿它来做模板恐怕并不方便。如果有许多类似这样接受命名参数的子程序，可以另外创建一个辅助子程序，专门用来完成参数解析转换工作。比如下面这个以匿名散列技术实现的子程序：

```
sub do_params {
  my ( $arg, $default ) = @_;
  my %param = @$default;
  my %input;

  if ( ref $$arg[0] eq 'HASH' ) {
    # 如果是命名参数，转换后存入散列
    %input = %{ $$arg[0] };
  }
  else {
    # 如果是占位参数，逐一命名后存入散列
    %input =
      map { $$default[ $_ * 2 ], $$arg[$_] } 0 .. $#_;
  }
  # 覆盖默认值
  @param{ keys %input } = values %input;
  \%param;
}
```

下面是它的用法：

```
sub uses_anon_hash_params {

  # 指向参数列表和默认值的引用
  my $param = do_params \@_,
    [ foo => 'val1', bar => 'val2' ];

  # 子程序 do_params 返回一个散列引用
  # 所以现在可以用诸如 $$param{foo}、$$param{bar} 等形式，比如：
  for ( keys %$param ) {
    print "$_: $$param{$_}\n";
  }
}
```

这里展示的技术各有优缺点，请选择适合实际应用的技术。如果都不合适，那必要的话自己选择一种改写一下好了。

www.TopSage.com

要点

- 通过散列给子程序传递命名参数。
- 通过合并散列设置参数默认值。
- 占位参数或命名参数，选择其中一种作为惯用手法。

条款48　通过参数原型声明以特殊方式解析参数

Perl支持子程序参数原型声明，但所声明的参数并非命名式的，并非严格类型化的，这和其他编程语言不同。不仅如此，Perl的这种特性实际是一种允许程序员编写类似内置函数那样使用参数的机制，不要把 Perl 的参数原型声明当作是数据类型验证的一种方式，它们只不过是提示Perl应该如何解析代码而已。

1. 自己编写 pop 函数

现在让我们来考虑一下该如何实现一个 pop2 函数，它应该可以删除并返回数组的最后两个元素。用起来就像是用内置的 pop 一样：

```
my @a    = 1 .. 10;
my $item = pop @a;
```

因为是返回两个元素，所以用起来应该像下面这样：

```
my ( $item1, $item2 ) = pop2 @a;
```

一般而言，如果想要实现像 pop2 这样的函数，就得用参数引用的方式，以便修改原始参数内容（见条款46）：

```
sub pop2_ref { splice @{ $_[0] }, -2, 2 }
```

但这样的话，我们就必须在使用时给出原始数组的引用，而不是直接给出数组变量：

```
my @a = 1 .. 10;
my ( $item1, $item2 ) = pop2_ref \@a;
```

现在我们可以引入参数原型声明了，通过对参数列表做一些特殊处理，实现类似内置函数pop一样的功能。

原型可以写在子程序声明或是参数定义的地方：

```
sub pop2 (\@) { splice @{ $_[0] }, -2, 2 }
```

原型是由原型原子构成的。原型原子是一些字符，有时会以反斜杠开头表明子程序所接受的参数类型。在这个例子中，\@ 这个原子代表 pop2 子程序需要单个命名数组作为参数。以反斜杠开头的原子，如 \$ 或 \@ 告诉 Perl 传递该参数的引用，所以 pop2 后面的数组会被取引用后传入，而不是作为一个包含多个值的列表整体传入。

原型还涉及对参数类型和数量是否合适的检查。比如对于数组以外的参数，pop2 函数就不会工作：

```
my @popped = pop2 %hash;
```

www.TopSage.com

运行这段代码的结果是一个编译时错误，因为 Perl 知道那个散列不属于这儿：

```
Type of arg 1 to main::pop2 must be array ↵
  (not private hash)
```

表 4-1 展示了我们可以在原型声明中使用的原子字符。

表4-1　子程序原型字符的类型与含义

原型字符	含　义
\$, \@, \%, \&, *	返回指向以 $、@、% 等开头的变量名或参数的引用
$	强制指定标量上下文
@, %	占用余下的参数，强制指定列表上下文
&	代码引用，如果是第一个参数，关键词sub可以省略
*	Typeglob
;	将必要参数和可选参数分隔开来

2. 多个数组参数

如果一个子程序需要接受两个数组参数并将它们“混合”进一个列表中该怎么办？比如从第一个数组中取一个元素，然后再从第二个数组中取一个元素，如此反复：

```
sub blend (\@\@) {
  local ( *a, *b ) = @_; # 比解引用要快
  my $n = $#a > $#b ? $#a : $#b;
  my @res;
  for my $i ( 0 .. $n ) {
    push @res, $a[$i], $b[$i];
  }

  # 也可以这么写：
  # map { $a[$_], $b[$_] } 0..$n;
  # 但 for 和 push 要更快一些
  @res;
}

# 使用举例
blend @a, @b;
blend @{ [ 1 .. 10 ] }, @{ [ 11 .. 20 ] };
```

采用类似的代码，我们还可以写一个像 foreach 那样遍历列表元素的子程序，但一次可以提取 n 个元素：

```
# for_n：同时提取 n 个元素遍历列表
sub for_n (&$@) {
  my ( $sub, $n, @list ) = @_;
  my $i;
  while ( $i < $#list ) {
    &$sub( @list[ $i .. ( $i + $n - 1 ) ] );
    $i += $n;
  }
}
```

www.TopSage.com

大家网 TopSage.com

```
# 使用举例
@a = 1 .. 10;
for_n { print "$_[0], $_[1]\n" } 2, @a;
```

如果代码要开放给全世界的话，使用类似\@和\%的符号时就要小心了，因为使用你所写代码的程序员也许不会意识到没用反斜杠的参数到子程序内部却是通过引用访问原始数据的。一定要这么做的话，请在文档中完整详细地描述。

另外，其实类似for_n的函数不用我们自己实现，因为List::MoreUtils模块的mesh和natatime早已帮我们做好了（见条款26）。

3. 要点

- 使用原型可以创建自己的数组或散列操作符。
- 使用原型可以创建接受独立数组作为参数的子程序。
- 避免过度使用原型，尤其是当它会给他人带来困惑的时候。

条款49　创建闭包锁住数据

在Perl中，**闭包**（closure）指的是可以包含能游离于作用域之外的词法变量的子程序，而这些变量数据之所以不会消失，并随同子程序引用一同保留在内存中，是因为子程序仍然拥有指向它们的引用。我们可以用闭包技术将数据限定在命名子程序或新匿名子程序中。

1. 命名子程序的私有数据

有时候我们的子程序需要一些只有它们自己能读取的数据。也就是说，对于任何数据，我们若想将它们的可见性限定在一个最小的可操控范围内，最简单的实现是将数据直接放在子程序内部：

```
sub some_sub {
  my $application_root = '/path/to/my/app';

  # 某些使用 $application_root 的操作
}
```

这么做的话，Perl在每次调用子程序时都得重建这个标量变量。如果我们并不需要修改该数据，那么这无疑就是浪费。也许它对性能影响不会很大，但从原理上来看，这终究是不够优雅和也不必要的做法。

我们可以在子程序外定义$application_root，不过要限定它的作用域在该子程序内，我们就可以将$application_root的定义和子程序打包在一个区块中。$application_root需要先于子程序定义，这样子程序就可以使用它，一般会放在BEGIN区块中打包：

```
BEGIN {
   my $application_root = '/path/to/my/app';

   sub some_sub {
```

www.TopSage.com

```
    ...;
  }
}
```

在 Perl 5.10 或更高版本中，我们可以通过 state 静态变量实现同样的效果，这个常见的模式现在也成为 Perl 的一个特性了。首次运行该子程序时，Perl 会定义 state 静态变量并赋值，而在随后的调用中，Perl 会忽略这行代码，该变量会保留前一次子程序运行时的值[①]：

```
use 5.010;

sub some_sub {
  state $application_root = '/path/to/my/app';

  # 用到$application_root 的某些操作
}
```

对于多次子程序调用，state 变量可以帮助维持变量原始取值：

```
use 5.010;

sub show_letter {
  state $letter = 'a';

  print "Letter is ", $letter++, "\n";
}

foreach ( 0 .. 5 ) {
  show_letter();
}
```

输出结果显示了$letter 的增长：

```
Letter is a
Letter is b
Letter is c
Letter is d
Letter is e
Letter is f
```

2. 子程序引用的私有变量

匿名闭包与使用 state 变量基本上是一回事情，但它的用处更大。采用闭包我们可以灵活自由地创建多个闭包，并按照特别的需求建立每个子程序。比如创建一个和前例功效相同的匿名闭包实现，大部分内容都是一样的，区别在于这是在运行时发生：

```
my $session = do {
  my $application_root = '/path/to/my/app';

  sub {
    ...;
  }
};
```

① 可以把它想象为只能一次写入的刻录盘。——译者注

www.TopSage.com

这种做法的妙处在于，我们可以动态创建满足当下需求的闭包子程序。尽管采用匿名子程序的形式好像看起来不错，但它实际和命名子程序没什么差别，一样缺乏灵活性。而像下面这样按照**工厂模式**（factory）动态创建闭包子程序的方式，显然要灵活得多：

```
my $session = closure_factory('/path/to/my/app');

sub closure_factory {
  my $application_root = shift;

  sub {
    ...;
  }
}
```

只要愿意，这样的闭包要多少有多少。比如用同样的工厂子程序（这个别致的名字指的是像工厂运用多条流水线一样动态创建子程序的子程序）创建独立计数的闭包子程序：

```
sub make_cycle {
  my ( $min, $max ) = @_;

  my @numbers = $min .. $max;
  my $cursor = 0;

  sub { $numbers[ $cursor++ % @numbers ] }
}

my $cycle_5_10 = make_cycle( 5, 9 );
my $cycle_f_m = make_cycle( 'f', 'm' );
```

当我们调用其中一个闭包时，它不会影响其他通过同一个工厂子程序创建的闭包：

```
foreach ( 0 .. 10 ) {
  print $cycle_5_10->(), $cycle_f_m->();
}
```

输出显示闭包的操作是相互独立的，但它们的输出却交织在一起：

```
5f6g7h8i9j5k6l7m8f9g5h
```

3. 闭包可以共享数据

我们不必将作用域外的数据限制在一个闭包中，只要创建子程序时把这些数据放在同一个作用域中，就能通过引用共享。比如 File::Find::Closures 模块，提供了一个能和 File::Find 一同工作的子程序 find，只要给它一个具体负责查找条件的子程序引用，它就能提取搜索结果：

```
use File::Find qw(find);
use File::Find::Closures qw(find_by_regex);

my ( $wanted, $reporter ) = find_by_regex(qr/*.pl/);

find( $wanted, @search_dirs );

my @files = $reporter->();
```

www.TopSage.com

这里的 find_by_regex 为我们处理了两个重要的细节，而每个细节都是由它自己的闭包完成的。首先，它创建 find 函数所需要的回调函数。在返回这两个闭包子程序之前，共同的作用域中又定义了数组变量@files，这样，回调函数运行后搜索到的文件就会保存到该数组中，随后提供的第二个闭包子程序，则用来访问这个文件列表：

```
# 取自 File::Find::Closures
sub find_by_regex {
  require File::Spec::Functions;
  require Carp;
  require UNIVERSAL;

  my $regex = shift;

  unless ( UNIVERSAL::isa( $regex, ref qr// ) ) {
    Carp::croak "Argument must be a regular expression";
  }

  my @files = ();

  sub {
    push @files,
      File::Spec::Functions::canonpath($File::Find::name)
      if m/$regex/;
  }, sub { wantarray ? @files : [@files] }
}
```

4. 要点

- 用词法变量创建子程序私有数据。
- 在 Perl 5.10 或更高版本中，处理私有数据可以使用 state 变量。
- 创建工厂子程序（也称生成器子程序）动态构建新的子程序。

条款 50　用子程序创建新子程序

如果经常要以相同参数调用某些固定的子程序，不妨创建一个新的子程序，由它负责记住这些参数。这被称为子程序的**柯里化**（Currying）。

比如下面这个子程序，根据给定模式找出数组中符合条件的元素，排序后返回：

```
sub my_sorted_grep {
  my ( $pattern, $array_ref ) = @_;

  my @results = sort grep /$pattern/o, @$array_ref;

  wantarray ? @results : scalar @results;
}
```

调用时，必须同时给出匹配模式和输入列表：

```
my @results = my_sort_grep qr/.../, \@input;
```

www.TopSage.com

这段代码并不长，但如果我们需要在代码中以同样的匹配模式做很多遍搜索呢？我们只能重复输入这些相同的代码，这就显然不够方便了。相同的匹配模式分散在代码各处，不如封装到子程序中，单点定义：

```
my $find_buster = sub {
  my ($array_ref) = shift;
  my_sorted_grep( qr/Buster/i, @$array_ref );
};
```

这样，调用时就不会那么繁琐了：

```
my @results = $find_buster->( \@input );
```

我们还可以根据旧函数创建新函数。Perl 的这类**高阶功能**（higher-order function）整合起来足以写一本书了，幸运的是，Mark Jason Dominus 已经写了 *Higher Order Perl*。

下面是一些对字符串做转换的简短子程序：

```
sub my_uc       { uc $_[0] }
sub my_ucfirst  { ucfirst $_[0] }
sub trim_front  { my $s = shift; $s =~ s/^\s+//; $s }
sub trim_back   { my $s = shift; $s =~ s/\s+$//; $s }
```

现在，给定一个字符串，如果想要将它开头和结尾的空白字符截掉，并将第一个字符转为大写，可以依次单独调用所有这些子程序，然后将结果返回原始标量变量：

```
my $string = '';
$string = trim_front($string);
$string = trim_back($string);
$string = my_ucfirst($string);
```

也可以跳过中间步骤，将上一步骤的结果作为下一步骤的输入，但这样写的话看起来会比较杂乱：

```
$string = my_ucfirst( trim_back( trim_front($string) ) );
```

其实，可以将这些函数并入一个组合子程序：

```
my $ucfirst_and_trim = sub {
    my $string = shift;
    my_ucfirst( trim_back( trim_front($string) ) );
};
```

现在，任何需要同时执行这三项操作的地方都可以使用新写的子程序：

```
$string = $ucfirst_and_trim->($string);
```

这和编写普通的子程序没有太多不同，无非是创建了一些可重用的代码，但区别在于，现在这种方式是在运行时实现的。

更进一步，我们可以写一个工厂子程序，用以动态创建新的组合式子程序。也许在运行之前我们无法确定哪些操作是需要的，所以就算用匿名子程序也同样不能解决问题。而组合式子程序可以接受一系列子程序引用，然后构造一个串联这些子程序运行结果的子程序并返回：

```
sub composer {
  my (@sub_refs) = @_;
```

www.TopSage.com

TopSage.com

```
    sub {
      my $string = shift;
      foreach my $sub_ref (@sub_refs) {
        $string = $sub_ref->($string);
      }
      return $string;
    };
}
```

这样我们就可以自由组合新的子程序了，比如，在上例中按照我们想要的顺序逐项转换：

```
my $ucfirst_and_trim =
  composer( \&trim_front, \&trim_back, \&my_ucfirst );

$string = $ucfirst_and_trim->($string);
```

组合子程序可以实现各种串联方式，不同的方式返回不同的子程序引用（或多个子程序引用）。

要点

- 根据程序运行状态可以动态创建新的子程序。
- 封装一组子程序，以默认参数联合运行。
- 用生成器子程序构建高阶函数。

第5章 文件与文件句柄

用Perl处理文件非常容易。经过多年积累，Perl已经有了各式强大的工具，处理数据也好，检查文件内容也好，读取再写回去也好，都有现成的，拿来就可以用，非常方便。

事实上，Perl的优势不仅仅在于处理文件。你可能认为文件无非是磁盘里有着漂亮图标的那些东西罢了。但Perl却能借用文件句柄接口处理几乎所有形式的数据。通过文件句柄我们能完成大部分重要的任务。文件句柄还可以保存为普通的标量变量，以便将来选择要操作的对象。

条款57 不要忽略文件测试操作符

最常听到Perl新手问的一个问题就是："怎样才能得到文件大小？"而其他新手的回答也总是一成不变，他们会给出啰嗦的办法，虽然能用，但要输入好多字符：

```
my (
  $dev,    $ino,     $mode, $nlink, $uid,
  $gid,    $rdev,    $size, $atime, $mtime,
  $ctime,  $blksize, $blocks
) = stat($filename);
```

或者，他们知道该怎样省去用不到的多余变量，而使用数组切片（见条款9）：

```
my ($size) = ( stat $filename )[7];
```

本来很普通的工作，如果写起来非常麻烦，就该停下来想想是否有其他简便易行的方式。Perl就是专为简化常见任务而设计的，所以这类任务理应有一个更为简捷的实现方式。实际上，用-s文件测试操作符，就能直接取得文件大小，单位是字节：

```
my $size = -s $filename;
```

许多人都忽视了Perl的文件测试操作符。可能是写C程序的时间太长了，也可能看到别人那么写就一直照搬，也可能他们怀疑这些操作符不靠谱。但不管怎样，说出去会让人笑话，放着言简意赅的文件测试操作符不用，偏要用stat写一堆复杂啰嗦的代码，多少有点舍近求远的意味，更何况文件测试操作效率更高，可读性也更好。说来也奇怪，文件测试操作符列在**perlfunc**文档中所有内建函数的最前面。这是因为按字面排序的话，像-x这样短划线开头的就排在前头，而所有文件测试操作符都归在-x名下。在perldoc中指定-x参数，就可以看到所有文件操作符

www.TopSage.com

的说明：

```
% perldoc -f -X
```

文件测试可以很自然地放在循环语句或条件判断语句中使用。下面的例子从指定目录取得所有文件列表，依次使用-T 检查每个文件是否为文本文件。实际上它会从文件内容取样，并作适当的推测，从而得出是否为文本文件的结论。

所有文件测试默认情况下都要使用变量$_：

```
my @textfiles = grep { -T } glob "$dir_name/*";
```

-M 和-A 这两个文件测试会返回文件最后修改和访问以来的天数，但计算的时候以 Perl 程序启动的时间作为标准。换句话说，用程序启动时的时间减去最后修改和访问的时间，得到的就是相差的天数。正数说明修改和访问发生在程序启动之前，负数说明发生在程序启动之后。可能看起来有些古怪，但却比较符合人类表述年代的思维方式。比如，要找出过去七天里不曾更新过的文件，可以限定-M 返回大于 7 的那些文件：

```
my $old_files = grep { -M > 7 } glob '*';
```

同样，要找出程序启动后有更新的文件，只需返回-M 是负数的那些即可。请看下面的例子，如果-M 返回小于零的值，则通过 map 返回匿名数组，数组元素为文件名和该文件自上次被修改以来的天数。否则只返回空列表：

```
my @new_files = map { -M < 0 ? [ $_, -M ] : () } glob '*';
```

1. 复用测试结果

如果想找出所有人为当前运行程序的用户，并且权限为可执行的文件，可以在 grep 中联合使用多个文件测试：

```
my @my_executables = grep { -o and -x } glob '*';
```

实际上，文件测试操作在幕后调用的就是 stat 函数，拿到我们所要的结果便返回。每次运行文件测试，Perl 都会重新调用一次 stat。也就是说，上面的例子中 Perl 对$_调用了两次 stat。

如果要对同一个文件做多次文件测试操作，可以使用虚拟文件句柄 _（就是一个下划线字符）以节约不必要的开销。它会告诉文件测试操作符，不必再调用 stat 了，直接用上次测试的结果。只要简单地把 _ 放在要做的文件测试后面，针对每个文件就只会调用一次 stat 函数了：

```
my @my_executables = grep { -o and -x _ } glob '*';
```

2. 栈式文件测试

从 Perl 5.10 开始，已经可以使用栈式文件测试了。也就是说，对同一个文件或文件句柄，可以同时进行多项属性测试。比如测试那些当前用户能读能写的文件，只要在文件名之前同时列出 -r 和 -w 即可：

```
use 5.010;

if ( -r -w $file ) {
  print "File is readable and writable\n";
```

www.TopSage.com

```
}
```

其实幕后并没有什么神奇的魔法，不过是原来分开写法的一种新式语法糖，走走捷径罢了。观察下面的老式写法，这是同上面最为接近的等效写法，先作 -w 测试，再作 -r 测试，并要求同时成立：

```
if ( -w $file and -r $file ) {
  print "File is readable and writable\n";
}
```

由此重写上一节的例子，可以简化为：

```
my @my_executables = grep { -o -x } glob '*';
```

3. 要点

- 文件测试操作符会做 stat 运算，所以不必直接调用它。
- 用虚拟文件句柄 _ 复用之前一次 stat 的测试结果。
- 栈式文件测试操作可用于 Perl 5.10 及后续版本。

条款 52　始终以三项参数的形式调用 open

很久以前，在 Perl 最古旧的版本中，为了创建一个文件句柄，我们必须同时指定文件打开模式和文件名：

```
open( FILE, '> output.txt' ) || die ...; # 过时且不恰当
```

这样的写法好像没什么问题，但若改用变量的话，就不好说了：

```
open( FILE, $read_file ) || die ...; # 有风险，也过时了
```

变量 $read_file 中的内容可以做两件事，一是指定打开模式，二是指定打开的文件。如果有人存心要戏弄你，将文件名以 > 开头，那数据就悲剧了：

```
$read_file = '> birdie.txt'; # 永别了，birdie!
```

open 函数两个参数的形式有一种特别的处理方式，看到重定向字符的话，就认为要写一个文件。虽然初衷是要讨巧，但其实并不讨好，这种附带歧义的特性会让代码遭受攻击，也容易出现意外。

想想看，如果有人要破坏你的文件，只需耍点小花招，比如，原来是以只读方式打开的文件，仅改动其文件名，它就会以读写模式工作，就能轻易覆盖掉原来的内容：

```
$read_file = '+> important.txt';
```

更为邪恶的是，若他们调用管道， open 就会运行外部命令：

```
$read_file = 'rm -rf / |'; # 看起来都叫人觉得心疼!
```

而正当你觉得万事俱备、一切顺利的时候，骇客在凌晨三点跑出来，趁你在床上呼呼大睡，偷偷启动外部命令，其结果可想而知。

从 Perl 5.6 开始，已有了解决这种问题的办法，就是使用三个参数形式的 open。而且从现在

www.TopSage.com

开始，就应该养成使用这种形式的习惯。

需要读取文件时，就明确告诉它只能从文件读：

```
open my ($fh), '<', $read_file or die ...;
```

文件名不再拥有双层含义，它就是纯粹的文件名，所以混淆原本意图的概率被大大降低了。$read_file 变量中任何一个字符都没有特别含义，任何原本表示重定向、管道等特殊意义的字符现在都只是纯粹的文字。

同样，要写数据到文件时，也必须明确是覆盖还是追加模式：

```
open my ($fh), '>',  $write_file  or die ...;
open my ($fh), '>>', $append_file or die ...;
```

两个参数形式的 open 无法处理文件名中包含空格的文件。因为它会尝试对给定的字串分解为模式和文件名两部分，遇到空格也就自然而然地忽略掉。所以万一你需要文件名首末两端均保留空格，用两个参数就没法正确处理了。而改用三个参数的形式，就不存在这个问题。文件名当中可以保留任何白字符。有时间可以试试看，创建一个名字以换行符开头的文件。可以工作吗？很好。之后我们会教你怎么删除它。

要点

- 始终记得使用三个参数形式的 open。
- 用词法标量存储文件句柄的引用。
- 用 or 结构检查 open 操作是否成功，避免调用无效句柄。

条款 53　采用不同方式读取数据流

一般我们用行输入操作符 <> 读取数据流，如果是标量上下文，就返回一行，如果是列表上下文，就返回数据流中所有的数据。究竟用哪个，取决于实际应用对效率的要求、对每行数据的访问方式以及各种其他因素（比如语法上写起来是否方便）。

总体而言，一次读取一行的方式在时间和内存开销上效率是最高的。而 while (<>) 这种隐式的写法，在速度上和相对应的显式写法是一样的：

```
open my ($fh), '<', $file or die;

while (<$fh>) {
  # 对当前行$_做某些处理
}
while ( defined( my $line = <$fh> ) ) { # 显式完整写法
    # 对当前行$line 做某些处理
}
```

请注意第二个循环中的 defined 操作符。当读到文件最后一行，如果只有字符 0 并且没有换行符的情况，它会返回 undef，从而避免出现提前结束循环并跳过最后这行的情况。虽说一般不会出现这种状况，但小心处理总是没错的。

也可以在 foreach 循环中使用类似的语法读取整个文件内容到内存，只需一步：

www.TopSage.com

大家网 TopSage.com

```
foreach (<$fh>) {
  # 对当前行 $_ 做某些处理
}
```

一次全部读入的方式自然要比一次一行的方式耗费内存。如果只是要逐行走一遍，一次一行就够了。对于短小文件来讲，两者在性能上的差异并不显著，所以不必过于在意。

一次全部读入也有其优势，尤其是当需要对文件进行诸如排序这类操作时：

```
print sort <$fh>; # 打印排序后的每一行
```

如果需要同时访问多行内容，一次全部读入的方式就不可或缺。比如根据之前或之后行的内容，决定当前行的处理方式，就必须同时取得这些行的信息。下面的例子，在找到某个包含 Shazam 的行时，会将该行连同相临的上下两行一同打印出来：

```
my @f = <$fh>;
foreach ( 0 .. $#f ) {
  if ( $f[$_] =~ /\bShazam\b/ ) {
    my $lo = ( $_ > 0 )    ? $_ - 1 : $_;
    my $hi = ( $_ < $#f ) ? $_ + 1 : $_;
    print map { "$_: $f[$_]" } $lo .. $hi;
  }
}
```

当然也可以改写成一次一行的方式，不过这样写出的代码明显要复杂得多：

```
my @fh;
@f[ 0 .. 2 ] = ("\n") x 3;
for ( ; ; ) {
  # 用数组切片赋值的方式构造临时队列
  @f[ 0 .. 2 ] = ( @f[ 1, 2 ], scalar(<$fh>) );
  last if not defined $f[1];
  if ( $f[1] =~ /\bShazam\b/ ) { # …… 查找 Shazam
    print map { ( $_ + $. - 1 ) . ": $f[$_]" } 0 .. 2;
  }
}
```

用数组切片赋值的方式手工维护一个相关行的队列，比等价的一次全部读入方式要慢得多，但对于庞大数据量的处理它却不失为上上策。当然，队列的操作也可以采用数组索引而不必每次赋值，虽然代码形式上会复杂些，但速度更快。

1. 文件 Slurp

有时候我们的需求很简单，只是想尽可能快地一次读取所有文件内容，也许你会考虑将每行的分隔符都去掉后再将它们当成一个字符串读入。比如下面的代码，它比上面提到的任何一种处理方式都要快：

```
my $contents = do {
  local $/;
  open my ($fh1), '<', $file1 or die;
  <$fh>;
};
```

也可以选用 File::Slurp 模块替我们完成，只需一条函数，便可把全部文件内容读入标量

或者按行保存到数组中：

```
use File::Slurp;

my $text = read_file('filename');
my @lines = read_file('filename');
```

2. 用 read 或 sysread 获得最快速度

最后提一下 read 和 sysread 这两个操作符，如果对行边界无所谓，倒是可以用它们来实现快速扫描：

```
open my ($fh1), '<', $file1 or die;
open my ($fh2), '<', $file2 or die;

my $chunk = 4096; # 每次读取的数据块大小
my ( $bytes, $buf1, $buf2, $diff );

CHUNK: while ( $bytes = sysread $fh1, $buf1, $chunk ) {
  sysread $fh2, $buf2, $chunk;
  $diff++, last CHUNK if $buf1 ne $buf2;
}

print "$file1 and $file2 differ" if $diff;
```

3. 要点

- 如非必需，请避免一次读入完整文件内容。
- 用 File::Slurp 模块快速读取整个文件内容。
- 快速扫描读取文件可以使用 read 或 sysread。

条款 54　处理字符串的文件句柄

从 Perl 5.6 开始，我们可以在字符串上打开一个文件句柄，操作起来和普通文件、套接字或管道没什么两样。其实抛开句柄操作的是字符串这个事实，敞开思路去看的话，会发现获取和推送数据的方法也可以多种多样。使用句柄减少细枝末节的处理，一定程度上也会减少应用程序的复杂度，提高代码的可维护性。

并且，这样的变化不光影响到你一个人。虽然表面上看，从字符串打开文件句柄算不上什么特性，但它的确就是。因为你永远都无法预期将来倒腾你代码的人会怎样改动它，而保持操作上的简单和一致，就能在一定程度上避免产生混乱。

1. 从字符串读

对于多行字符串，不用拿正则表达式切分各行。只要在该字串标量的引用上打开一个文件句柄，然后像以往那样从该句柄读取数据即可：

```
my $string = <<'MULTILINE';
Buster
```

```
Mimi
Roscoe
MULTILINE

open my ($str_fh), '<', \$string;

my @end_in_vowels = grep /[aeiou]$/, <$str_fh>;
```

以后，如果想修改代码，让它从外部文件而不是源代码中的字符串读取内容，只需修改文件句柄名字就行，其余代码运行照旧：

```
my @end_in_vowels = grep /[aeiou]$/, <$other_fh>;
```

如果把中间对数据的实际操作抽出来，写成一个子程序的话，代码的编写和逻辑上的意义会变得更清晰，也更灵活。子程序不管要处理的数据来自何方，只要能从文件句柄拿到数据就行：

```
my @matches = ends_in_vowel($str_fh);
push @matches, ends_in_vowel($file_fh);
push @matches, ends_in_vowel($socket);

sub ends_in_vowel {
  my ($fh) = @_;

  grep /[aeiou]$/, <$fh>;
}
```

2. 写入字符串

既然能读，那么通过句柄写数据到字符串也没问题，无非是打开句柄的模式不同罢了：

```
my $string = q{};

open my ($str_fh), '>', \$string;

print $str_fh "This goes into the string\n";
```

同样，追加数据到字符串末尾也一样简单：

```
my $string = q{};

open my ($str_fh), '>>', \$string;

print $str_fh "This goes at the end of the string\n";
```

上面的代码可以略微简化一下，使用标量引用之前所做的声明，完全可以与在该引用被使用的行合并为一行。第一次看到这种写法或许会觉得奇怪，但在 Perl 里面这是很自然的事情：

```
open my ($str_fh), '>>', \my $string;

print $str_fh "This goes at the end of the string\n";
```

尤其是在子程序或者方法内部期望的是打印数据到文件句柄，而我们却希望将内容捕获至内存时，这种方式就显得特别方便。这样就不必为写入内容之后再读出而创建一个新文件，我们只需要直接捕获内容就好了。

www.TopSage.com

3. seek 与 tell

一旦拿到字符串句柄，就可以适用所有针对文件句柄的操作，包括在这个“虚拟文件”内任意游走。用它可以打开一个字符串以供读取，也可以移到某个位置或者读取一定数量的字节。这对于图像文件或是其他二进制（并非以行为单位组织内容的）文件的处理，是极其有用的：

```
use Fcntl qw(:seek); # 为了引入一些常量
my $string = 'abcdefghijklmnopqrstuvwxyz';

my $buffer;
open my ($str_fh), '<', \$string;

seek( $str_fh, 10, SEEK_SET ); # 从开头处移至 10 个字节后的位置
my $read = read( $str_fh, $buffer, 4 );
print "I read [$buffer]\n";
print "Now I am at position ", tell($str_fh), "\n";

seek( $str_fh, -7, SEEK_CUR ); # 往回走 7 个字节
my $read = read( $str_fh, $buffer, 4 );
print "I read [$buffer]\n";
print "Now I am at position ", tell($str_fh), "\n";
```

输出结果表明我们在字串内容间可以前后游走：

```
I read [klmn]
Now I am at position 14
I read [hijk]
Now I am at position 11
```

如果打开时选用可读可写模式，即指定为 +< ，便可以替换特定位置上的字串内容：

```
use Fcntl qw(:seek); # 为了引入一些常量
my $string = 'abcdefghijklmnopqrstuvwxyz';

my $buffer;
open my ($str_fh), '+<', \$string;

# 从开头处移至 10 个字节后的位置
seek( $str_fh, 10, SEEK_CUR );
print $str_fh '***';
print "String is now:\n\t$string\n";

read( $str_fh, $buffer, 3 );
print "I read [$buffer], and am now at ",
  tell($str_fh), "\n";
```

输出结果说明字串内容已被修改，修改后的字串仍然可以继续读取：

```
String is now:
    abcdefghij***nopqrstuvwxyz
I read [nop], and am now at 16
```

用 substr 也能做相同的操作，但它的操作对象仅限于字符串。而用句柄处理，不仅操作灵活，能做的事也比 substr 更多。

www.TopSage.com

4. 要点

- 将字符串当作文件来处理，以避免某些特殊情况。
- 借助可读文件句柄，可以将字符串按行拆分。
- 借助可写文件句柄，从目标字符串捕获数据。

条款55　灵活的输出方式

使用硬编码（或默认）文件句柄写出来的程序，一方面灵活性不够，另一方面也容易让用户感觉不爽。比如下面这样的写法：

```
print "This goes to standard output\n";
print STDOUT "This goes to standard output too\n";
print STDERR "This goes to standard error\n";
```

这样的语句放在程序中，无疑会降低代码的灵活性。为了迎合它们，人们必须使出浑身解数，像玩杂耍般反复倒腾不同句柄。他们本不该负责局部化任何文件句柄，也无需为了改变输出方向而重新定义标准文件句柄。尽管道理没错，但人们依旧习惯这么写，只是因为这种写法快速简单，又很直观，而且他们也并不知道更好的做法其实同样也很简单。

具体的办法很多，但最方便的还是采用面向对象的设计。如果需要输出数据，可用调用对象的方法获取输出文件句柄，比如在下例中调用 get_output_fh，由它负责数据输出的方向：

```
sub output_method {
  my ( $self, @args ) = @_;

  my $output_fh = $self->get_output_fh;

  print $output_fh @args;
}
```

为了实现这种方式，我们需要一个方法来设置输出文件句柄。这可以通过一套常规的对象数据存取器方法来实现。如果我们没有做任何设置，get_output_fh 会返回默认的 STDOUT：

```
sub get_output_fh {
  my ($self) = @_;

  return $self->{output_fh} || *STDOUT{IO};
}

sub set_output_fh {
  my ( $self, $fh ) = @_ ;

  $self->{output_fh} = $fh;
}
```

有了这样的访问接口，别的程序员就能按照需要灵活地设置和改变数据输出的流向：

```
$obj->output_method("Hello stdout!\n");

# 输出到字符串
```

www.TopSage.com

TopSage.com

```
open my ($str_fh), '>', \$string;
$obj->set_output_fh($str_fh);
$obj->output_method("Hello string!\n");

# 通过网络发送数据
socket( my ($socket), ... );
$obj->set_output_fh($socket);
$obj->output_method("Hello socket!\n");

# 保存到字符串的同时，输出到 STDOUT
use IO::Tee;
my $tee =
  IO::Tee->new( $str_fh, *STDOUT{IO} );
$obj->set_output_fh($tee);
$obj->output_method("Hello all of you!\n");

# 将数据丢弃
use IO::Null;
my $null_fh = IO::Null->new;
$obj->set_output_fh($null_fh);
$obj->output_method("Hello? Anyone there?\n");

# 运行时决定：用户交互模式下输出到 stdout，非交互模式下使用空句柄屏蔽输出
use IO::Interactive;
$obj->set_output_fh( interactive() );
$obj->output_method("Hello, maybe!\n");
```

不止如此，这种做法还有许多连带的好处。想要添加一个将输出结果按字符串形式返回的方法吗？没有问题！实际上代码已经在那里了，只需临时改换一下输出句柄，将输出内容存到字符串（见条款 54）再返回即可：

```
sub as_string {
  my ( $self, @args ) = @_;

  my $string = '';
  open my ($str_fh), '>', \$string;
  my $old_fh = $self->get_output_fh;
  $self->set_output_fh($str_fh);
  $self->output_method(@args);

  # 恢复之前设置的文件句柄 fh
  $self->set_output_fh($old_fh);

  $string;
}
```

现在如果想要关闭所有输出，做法也很干脆，只需设置使用空句柄即可：

```
$obj->set_output_fh( IO::Null->new )
  if $config->{be_quiet};
```

要点

- 为了灵活性，操作文件句柄时不要用硬编码。

- 给其他程序员修改输出句柄的自由。
- 用 IO::Interactive 模块检查是否处于用户交互模式。

条款56　用 File::Spec 或 Path::Class 处理文件路径

Perl 能够运行于几百种不同的操作系统平台上，因此软件开发工程师将具备良好的可移植性视作编程的一条铁律。无论是我们最喜欢的系统，还是我们最不情愿用的系统，优秀的程序都必须能在其中流畅运行。这就涉及文件路径的处理。为保证良好的可移植性，底层兼容细节的处理可以交由以上两个模块之一负责，不光代码更安全，使用起来也轻松简便。

1. 用 File::Spec 提高可移植性

从 Perl 诞生时，File::Spec 模块就一起发布了，其最简易的用法是直接调用它的函数。在加载该模块时，会自动导入若干常用函数到当前名字空间：

```
use File::Spec::Functions;
```

要构建新的文件路径，需要磁盘卷（有时不需要）、目录以及文件名这些元素。磁盘卷和文件名的构建很容易：

```
my $volume = 'C:';
my $file   = 'perl.exe';
```

而一步一步构建目录则需要花点力气，但也不是太难。由 rootdir 函数开头，逐个列出文件所在的目录层次，然后用 catdir 串接起来，所连接的目录间隔字符则根据当前特定的操作系统而定：

```
my $directory =
  catdir( rootdir(), qw(strawberry perl bin) );
```

习惯于 Windows 或 UNIX 的人，可能就不太适应像 VMS 这样的系统，因为这类系统都是以相同的格式来构造目录和文件名中各部分的信息。所以借助 File::Spec，就不用考虑底层的这些细节，只需关注在程序的业务逻辑上就好了。

现在我们得到了文件路径的三个部分，用 catpath 组合起来：

```
my $full_path =
  catpath( $volume, $directory, $file );
```

在类 UNIX 系统中，catpath 会忽略表示磁盘卷的参数。所以如果没有这部分的话，可以用 undef 替代：

```
my $full_path =
  catpath( undef, $directory, $file );
```

如果我们认为这个程序将来只可能在本地系统上运行，还坚持这么写就好像有点傻。其实想要偷懒很简单，和谁都别提这个程序，没人知道，自然也就不必花力气移植，多好啊。可谁能保证呢？一开始就预留下不是更好吗？

File::Spec 还有许多其他函数，可以对路径进行分拆组合，也可以取得本地系统中表示上

www.TopSage.com

级目录、临时目录和空设备等的路径。

2. 尽可能选用 Path::Class

基于 File::Spec 封装而来的 Path::Class 模块，为常见的路径操作提供了更为便捷的方法。它把对路径的拼装逻辑解放到对路径的使用逻辑上来，免去了一大堆琐碎的细节。作为开始，先构造一个表示文件或目录的对象。在 Windows 上，直接将 Windows 下的文件路径给 file 函数即可，它会理解并做好一切。file 函数默认假设给定的路径位于本地文件系统上：

```
use Path::Class qw(file dir);

my $file = file('C:/strawberry/perl/bin/perl.exe');
```

该文件并不需要真实存在，$file 对象不会对路径做任何真实性验证，它仅仅是按照文件系统规范构造这样一条路径而已。

如果不是在 Windows 系统上运行程序，但还是需要以 Windows 上的路径工作，可以选用 foreign_file 代替：

```
my $file = foreign_file( 'Win32',
  'C:/strawberry/perl/bin/perl.exe' );
```

现在$file 可以完全正确地按照 Windows 文件系统规范处理该路径了。如果我们需要换种方式并将它转换为适合其他系统的路径，则可以使用 as_foreign 方法：

```
# /strawberry/perl
my $unix_path = $file->as_foreign('Unix');
```

得到表示文件的对象之后，对文件的操作无非就是调用该对象。

要取得文件句柄读取数据，可调用无参数的 open。它实际上是对 IO::File 的封装，所以相当于调用了 IO::File->new：

```
my $read_fh = $file->open
  or die "Could not open $file: $!";
```

要创建一个新的文件，第一步自然还是创建 file 对象。当然，构造对象时仅仅处理路径信息，并不会实际创建文件，只有当我们用 > 作为参数调用 open 时，文件才会写到系统上，并由此返回给你一个可写的文件句柄用于后续处理：

```
my $file = file('new_file');

my $fh = $file->open('>');

print $fh "Put this line in the file\n";
```

我们可以从现有的文件对象取得上级目录对象，然后打开目录句柄：

```
my $dir = $file->dir;
my $dh = $dir->open or die "Could not open $dir: $!";
```

对于现有的目录对象，也可以直接得到它的上级目录对象：

```
my $parent = $dir->parent;
```

www.TopSage.com

用 readdir 函数可以从目录句柄依次读取目录中的所有文件。但每次调用只返回文件名，而不是完整的路径，所以需要的话可以另外补上。这不难，交给 file 函数拼接就成：

```
while ( my $filename = readdir($dh) ) {
  next if $filename =~ /^\.\.?$/;
  my $file = file( $dir, $file );
  print "Found $file\n";
}
```

3. 要点

- 不要用硬编码处理文件路径中有关操作系统的特定细节。
- 用 File::Spec 或 Path::Class 构造和处理具备可移植性的路径。

条款57　将数据留于磁盘以节约内存

现今的数据集往往异常庞大。不管是分析 DNA 序列，还是解析博客文章，所要处理的数据总量很容易就会超过程序允许的内存大小。Perl 程序员处理超大数据集时经常会见到“Out of memory!”这样的错误提示。

当然，要解决这个问题，也不是没有办法。最简单的，先确认下我们的程序进程允许使用多少内存空间。如果允许，就让操作系统多分配些内存资源给它。

不过单纯地增加内存终归是治标不治本。若是数据不断增长，很快又会陷入原来的困境。

此外，有一些对策是用以减少不必要的内存开销，接下来我们逐一介绍。

1. 逐行读取文件

第一个也是最显而易见的对策，就是逐行读取。其实本来就没必要一次性加载所有文件内容到内存的。我们可以将整个文件逐行读入一个数组：

```
open my ($fh), '<', $file or die;
my @lines = <$fh>;
```

然而，如果你实际上并不同时需要所有的数据，那么按照需要读取适量的数据进行处理即可：

```
open my ($fh), '<', $file or die;

while (<$fh>) {
  # ……对当前行处理
}
```

2. 将大的散列表保存到 DBM 文件

有这样一种常见的情况，有时候我们需要根据某些关键字查找对应关联的一堆数据，而当这样的关键字不计其数时，每次查找数据都加到内存循环一遍，无疑会对内存开销提出挑战。比如查找日志，每条日志都有一个对应 ID，以及记录日志的时间和日志内容。日积月累，这样的日志数不胜数。我们可以在查询之前，先把数据加载到 DBM 文件，相当于是一个关键字为日志 ID 的散列表。这么一来，所做的查询操作就从内存搬到了硬盘上，大大节约了内存。如下例中的

www.TopSage.com

build_lookup 函数，运行起来好像数据都在内存，但实际它们却是保存在外部的数据库文件，我们仅仅是将该文件绑定进而通过散列表的方式进行访问而已：

```
use Fcntl; # 为了引入 O_RDWR、O_CREAT 等常量

my ( $lookup_file, $data_file ) = @ARGV;

my $lookup = build_lookup($lookup_file);

open my ($data_fh), '<', $data_file or die;

while (<$data_fh>) {
  chomp;
  my @row = split;
  if ( exists $lookup->{ $row[0] } ) {
    print "@row\n";
  }
}

sub build_lookup {
  my ($file) = @_;
  open my ($lookup_fh), '<', $lookup_file or die;

  require SDBM_File;
  tie( my %lookup, 'SDBM_File', "lookup.$$",
    O_RDWR | O_CREAT, 0666 )
    or die
    "Couldn't tie SDBM file 'filename': $!; aborting";

  while (<$lookup_file_handle>) {
    chomp;
    my ( $key, $value ) = split;
    $lookup{$key} = $value;
  }

  return \%lookup;
}
```

生成查询用的数据库文件可能会比较耗时，所以应该尽可能减少重复生成的次数。如果可以，预先生成 DBM 文件，在运行时才打开。一次生成，多次使用，而其他程序也能共享。

SDBM_File 是用 Perl 实现的 DBM 数据库，但它的灵活性并不是很好。如果你的系统支持 NDBM_File 或者 GDBM_File，可用它们替代。

3. 把文件当作数组来读取

如果觉得基于关键字查找的方式不够灵活，可以试试 Tie::File 模块，它将文件每一行当作数组元素来处理，但并不会全部加载到内存。我们可以在文件内采用导航处理，就好像操作普通数组一般；也可以在任何时候访问文件中的任何一行，像下面随机有奖打印的程序一样：

```
use Tie::File;

tie my @fortunes, 'Tie::File', $fortune_file
  or die "Unable to tie $fortune_file";
```

www.TopSage.com

大家网 TopSage.com

```
foreach ( 1 .. 10 ) {
  print $fortunes[ rand @fortunes ];
}
```

4. 使用临时文件和临时目录

如果没有预先准备好的文件，任何时候我们都可以自己写一个临时文件应急。File::Temp模块会自动创建一个名字唯一的临时文件，并在使用之后自动清除。这种方式特别适合一次性使用的情况，比如对某个文件创建新的版本，可以先写一个临时文件，等全部内容更新完毕后，再重命名覆盖掉原来版本：

```
use File::Temp qw(tempfile);

my ( $fh, $file_name ) = tempfile();

while (<>) {
  print {$fh} uc $_;
}

$fh->close;

rename $file_name => $final_name;
```

File::Temp甚至还可以创建临时目录，存放一堆临时文件。比如从网上下载一堆Web页面，存到临时目录以便后续处理：

```
use File::Temp qw(tempdir);
use File::Spec::Functions;
use LWP::Simple qw(getstore);

my ($temp_dir) = tempdir( CLEANUP => 1 );

my %searches = (
  google    => 'http://www.google.com/#hl=en&q=perl',
  yahoo     => 'http://search.yahoo.com/search?p=perl',
  microsoft => 'http://www.bing.com/search?q=perl',
);

foreach my $search ( keys %searches ) {
  getstore( $searches{$search},
    catfile( $temp_dir, $search ) ) );
}
```

关于File::Temp有个忠告：它默认按照二进制模式打开文件。所以如果你需要按照行来处理文件，或者以不同的编码方式打开（见条款73），就必须自己设定文件句柄的binmode。

5. 要点

- 将大的散列表存为磁盘上的DBM文件，以节省内存。
- 用Tie::File模块把文件当作数组来处理。
- 用File::Temp模块创建临时文件和目录。

www.TopSage.com

第6章 引 用

从Perl 5才开始引入的引用是Perl学习中的一个重点，它为复杂数据结构和面向对象的编程方式打开了一扇门。要想良好组织数据并对其进行整体传递，引用是关键。如果希望自己的Perl技艺更进一步，熟练掌握引用必不可少。

虽然本章介绍的只是一些关于引用的小技巧，但其中值得你学习的是如何高效创建并管理数据结构。要掌握这些技能，就需要不断练习。还等什么？现在就开始吧！

条款58 理解引用和引用的语法

引用实际是一个标量值，我们可以像数字和字符串一样，把引用直接存储在标量变量中或是作为元素存到数组或散列中。

我们可以把引用当成是指向其他Perl对象的“指针”。引用可以指向任何类型的对象，包括其他标量（甚至是引用）、数组、散列、子程序和符号表。

除了一般的指针行为，Perl的引用其实和C或C++中的引用没有太多共同点。我们只能针对已经存在的对象创建引用，也不能事后再修改已经创建的引用，比如改为指向某个数组的下一个元素之类的操作。

我们可以将引用转换成字符串或数字，但Perl没有内置的方法将字符串或数字再转回成引用。虽然从语法上来说引用和其他标量值相同，但引用“知道”自己所指向的对象类型是什么。此外，每个指向Perl对象的引用都会增加那个对象的引用计数，以防止该对象被Perl的垃圾回收器回收掉。

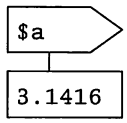
1. 创建引用

创建引用的方法有很多，最简单的就是用反斜杠操作符（或者说“取引用”操作符）作用于一个变量：

```
my $a           = 3.1416;
my $scalar_ref  = \$a;
```

反斜杠操作符会创建一个引用，指向它后面的变量所代表的值。如果用PEGS表示的话，看起来类似于：

www.TopSage.com



反斜杠操作符可以对任何类型的标量名称进行操作：

```
my $array_ref = \@a;
my $hash_ref  = \%a;
my $sub_ref   = \&a;
my $glob_ref  = \*a;
```

对于数组或散列元素，也一样可以取引用：

```
$array_elem_ref = \$a[0];
$hash_elem_ref  = \$a{'hello'};
```

它甚至可以操作直接量，尽管直接量的引用是只读的：

```
$one_ref  = \1;
$mode_ref = \oct('0755');
```

而对某个列表取引用的话，返回的是列表中每个元素的引用组成的新列表，而非列表本身的引用。它究竟返回何种数据的引用，得看具体的取引用方式。

来看下面这个 val 子程序：

```
sub val { return 1 .. 3 }
```

如果创建引用时用的是 & 符号，那么得到是指向子程序 val 的代码引用：

```
my $ref1 = \(&val);
```

如果创建引用前先调用 val，那么返回的是标量引用：

```
my $ref2 = \( val() );
```

由于上面这条表达式是在标量上下文中进行赋值操作的，所以最终得到的是 val 返回的列表中最后一个值的引用，也就是数字 3 的引用。

如果把赋值操作改为列表上下文，则会对应地赋值，最终得到的是一个标量引用，指向 val 返回的第一个值，也就是数字 1。

```
my ($ref3) = \( val() );
```

你也许会想，上面这种从子程序返回的列表中创建引用的方式，对包含直接量的列表也应该同样适用。是的，没错。下面这两个例子都会返回标量引用，指向数字 3：

```
my $ref4 = \( 1 .. 3 );

my $ref5 = \( 1, 2, 3 );
```

引用及其语法也可以变得很复杂，所以就算你自己清楚是怎么回事，其他接触你代码的人恐怕不会轻松理解，所以写代码时，应该尽可能保持意图清晰。

我们来看匿名数组的构造语法，它看起来和普通的列表很像，唯一不同的是它用圆括号而非方括号。它会在内存中创建一个没有名字的数组，并返回指向该数组的引用。这是创建一列数据

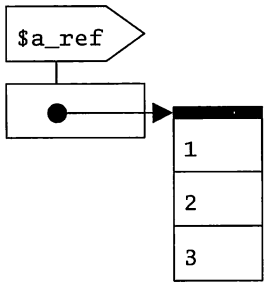
www.TopSage.com

的引用最自然最常用的方法。

```
my $a_ref = [ 1 .. 3 ];
```

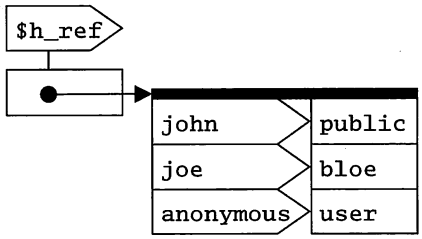
现在$a_ref 成了数组引用，指向一个包含数字 **1**、**2**、**3** 的匿名数组：

```
print ref $a_ref, " @$a_ref";
```



而类似的，匿名散列的构造语法使用花括号而非方括号，像下面这样：

```
my $h_ref = { anonymous => 'user' };
$h_ref->{'joe'}  = 'bloe';
$h_ref->{'john'} = 'public';
```



而定义子程序时如果没有给出名字，则返回匿名子程序引用。

有时我们也把子程序引用称为**代码引用**（coderef）。比如下面的代码，我们把代码引用存在标量变量$greetings 中，执行时通过引用呼叫代码：

```
my $greetings = sub { print "hello, world!\n" };
$greetings->();
```

这种匿名子程序的语法，常用于信号处理时挂接相应的处理逻辑，比如下面这个中断处理器：

```
$SIG{INTR} = sub { print "not yet--I'm busy\n" };
```

指向匿名子程序的引用其实非常有用（见条款 50）。从某种程度上来说，这和 C 语言中的函数指针很像。不过，由于匿名子程序的创建是运行时动态生成的，而非运行前完成的静态编译，所以它们会有各自特殊的属性，这一点和 LISP 有些相似。

在传递大字符串时，用标量引用就非常高效。比如下面的例子，第一句调用 some_sub()是把整个$string 的内容复制了一遍，而第二句调用只传递了该字符串的引用：

```
my $string = 'a' x 1_000_000;
```

www.TopSage.com

```
some_sub($string);

some_sub( \$string );
```

这个引用是通过“自动代入”（autovivification）的方式来实现的，我们将在本条款稍后部分继续讨论。

2. 使用引用

解引用（dereferencing）指的是取引用指向的那个值。解引用有许多不同形式的写法。

最规范安全的语法是将引用放在代码块（即花括号括起来的区块）中返回，然后把它当作某个变量或子程序的标识符使用。正如给出变量标识符，就能取得对应变量的值一样，通过写在花括号中的引用，我们可以取得该引用指向的数据。

来看下面这个例子，我们有一个普通标量变量 $a，随后创建了一个指向该标量的引用 $s_ref。之后便可以用代码块包裹引用的形式，像操作其他标量一样操作它：

```
my $a     = 1;
my $s_ref = \$a;
print ${$s_ref};
${$s_ref} += 1;
```

数组引用的工作方式大体相同。下面这个例子，有一个数组 @a，随后创建一个指向它的引用$a_ref。之后对数组的使用，不管哪种方式都能打印相同的数据，且任何一种写法修改数据的话，以另一种方式都可以看到内在数据的改变。

```
my @a     = 1 .. 5;
my $a_ref = \@a;
print "@a";
print "@{$a_ref}";
push @{$a_ref}, 6 .. 10;
```

其实代码块有多复杂都没关系，只要最后一条表达式求值返回的是引用的话就都一样。像下面这个例子，根据变量 $hi 的取值，选择不同数组并返回它的第三个元素值：

```
my $ref1 = [ 1 .. 5 ];
my $ref2 = [ 6 .. 10 ];
my $val = ${
  if ($hi) { $ref2; }
  else { $ref1 }
  }[2];
print $val; # 不是 3 就是 8
```

如果引用值存在标量变量中，我们可以省略花括号，直接使用标量变量的名字，前面再加上 $ 符号开头。如果它是指向引用的引用，我们可以用多个 $ 符号逐级解引用。比如下面的例子，用了一个指向标量的引用和一个指向引用的引用，分别打印原始变量所包含的单词 testing：

```
my $a         = 'testing';
my $s_ref     = \$a;
my $s_ref_ref = \$s_ref;
print "$$s_ref $$$s_ref_ref";
```

www.TopSage.com

TopSage.com

这种写法同样适用于散列引用：

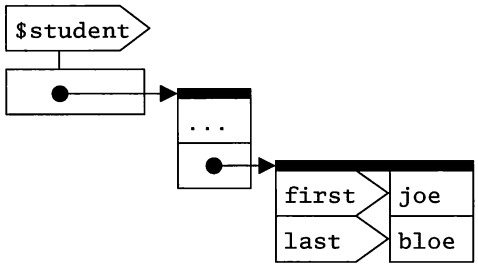
```
my $h_ref = { 'F' => 9, 'Cl' => 17, 'Br' => 35 };
print "The elements are ", join ' ', keys %$h_ref, "\n";
print "F's atomic number is is $$h_ref{'F'}\n";
```

类似 $$h_ref{'F'} 这样的表达式，或是更别扭的等效表达式 ${$h_ref}{'F'}，都使用得比较频繁。另外还有一种写起来稍微麻烦些，但看起来更直观的“箭头”语法，可以通过下标的方式取数组或散列的值：

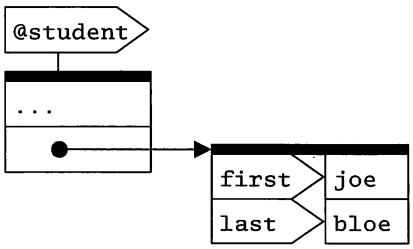
```
${$h_ref}{'F'} # 常规方式
$$h_ref{'F'}   # 标量变量方式
$h_ref->{'F'}  # 箭头方式
```

箭头可以级连。此外，如果箭头左右两边都是元素下标的话，那就可以省略箭头。下面两条 print 语句都会打印单词 joe：

```
$student->[1] = { 'first' => 'joe', 'last' => 'bloe' };
print $student->[1]->{'first'};
print $student->[1]{'first'};
```



不过，省略箭头和花括号时得小心，上面的例子中如果错误地省略了第一个箭头，那么就会把 $student[1] 当作数组 @student 的第二个元素，继而取这个元素（应该是散列引用）的键 first 对应的值。这显然和之前设定的数据结构不同：



最后提一点，不管是何种类型的引用，Perl 都会像处理标量一样处理它们——事实上从语法上来看，也确实没什么特别之处可以用以区分引用和普通标量。但引用其实还包含有目标对象的类型信息，我们可以用 ref 操作符获得该信息。

比如下面这个指向标量的引用，用 ref 打印出来的就是 SCALAR 类型：

```
my $s_ref = \1;
```

```
print ref $s_ref;
```

而匿名子程序的引用，打印出来的就是 **CODE**：

```
my $c_ref = sub { 'code!' };
print ref $c_ref;
```

3. 自动代入

如果我们把某个含有未定义值的标量当作指向其他数据类型的引用，Perl 就会自动创建相应类型的数据，并将这个标量作为指向该数据类型的引用。这种机制称为**自动代入**（autovivification）。比如下面这段代码就会自动创建一个包含四个元素的数组，并将 `$ref` 作为指向该数组的引用：

```
undef $ref;
$ref->[3] = 'four';
```

利用这个机制，我们可以免去逐层定义，非常方便地创建深层次的数据结构：

```
use Data::Dumper;

my $ds;

$ds->{top}[0]{cats}[1]{name} = 'Buster';

print Dumper($ds);
```

仅仅几行代码，我们就创建了一个层次比较复杂的数据结构：

```
$VAR1 = {
  'top' => [ {
    'cats' => [
      undef,
      {
        'name' => 'Buster'
      }
    ]
   } ]
};
```

条款 60 中展示了一个更长的关于自动代入的例子。

4. 软引用

如果对一个字符串值解引用，Perl 会返回以字符串为名字的变量的值。如该变量不存在，就会自动创建。这个称为**软引用**（soft reference）。

软引用可能是某次变量插值导致的结果：

```
my $str = 'pi';
${$str} = 3.1416;
print "pi = $pi\n";
```

我们甚至可以直接用字符串直接创建软引用：

```
${ 'e' . 'e' } = 2.7183;
print "ee = $ee\n"; # 2.7183
```

www.TopSage.com

类似这样的变量名可以不是合法的标识符，这意味着我们可以创建以**空白字符**（whitespace）甚至是**空字符**（null）命名的变量：

```
${}      = 'space';
${' '}   = 'space';
${'  '}  = 'two space';
${"\0"} = 'null';
```

注意，软引用是不会改变引用计数的。只有常规的“硬”引用才会增加引用计数。

打开 strict refs 编译指令可以禁用软引用（见条款 3），而且这么做有很好的理由：总还有其他更好的方式实现软引用所提供的功能。

5. 要点

- 可以通过引用操作符 \ 获取指向某个变量的引用，也可以用匿名数组构造语法 [] 和匿名散列构造语法 {} 创建引用。
- 通过解引用取得某个引用指向数据的值。
- 不要把普通标量当作引用使用，这样实际会创建一个软引用。

条款 59　将引用类型和原型进行比较

当我们把引用当成子程序参数进行传递时，多半需要验证一下参数类型是否符合预期。

```
sub count_matches {
  my ( $regex, $array_ref ) = @_;

  my $matches = grep /$regex/, @$array_ref;
}
```

如果给 count_matches 传递的参数类型不对，那后面的事情就都乱套了。原本希望拿到数组引用的，现在却传了完整的数组进去，Perl 是会报错的：

```
my $matches = count_matches( qr/.../, @array );
```

错误消息大体如下：

```
Not an ARRAY reference at ...
```

同样，如果第一个参数是个字符串，而非正则表达式引用，并且这个字符串碰巧是个非法正则表达式，那么同样会得到错误提示。比如字串里面的圆括号不成对：

```
my $matches = count_matches( '(...', \@array );
```

在子程序内部会把它当作正则表达式处理，结果发现不符合要求：

```
Unmatched ( in regex; marked by <-- HERE in ↵
  m/( <-- HERE / at ...
```

1. ref 操作符

想确定当前引用的类型，可以调用 ref 函数：

www.TopSage.com

```
my $array_ref = \@array;

my $type = ref $array_ref; # $type 的值是'ARRAY';
```

基本的引用类型无非是以下几种：SCALAR、ARRAY、HASH、CODE、GLOB 以及 Regexp。

2. 类型比较

有些人会把类型名字直接写死在代码里，然后和 ref 操作符的返回值作比较，类似这样：

```
sub count_matches {
  my ( $regex, $array_ref ) = @_;

  die "First argument needs to be a regex reference"
    unless ref $regex eq 'Regexp';
  die "Second argument needs to be an array reference"
    unless ref $regex eq 'ARRAY';

  my $matches = grep /$regex/, @$array_ref;
}
```

但这样将值限定死的做法很容易出错。比如正则表达式引用类型的名称，就和其他类型形式不一样，不光是混合了大小写，结尾还有个字母 p，常常有人乱猜乱用：

```
die '...' unless ref $regex eq 'REGEX'; # 错误
die '...' unless ref $regex eq 'Regex'; # 错误
```

其实我们大可不必记住这些类型名字的写法，只消通过简单的原型数据生成对应的类型名字，然后把它赋值给一个以大写字母命名的标量，权当常量就好了：

```
# 对空匿名数组使用 ref
my $ARRAY_TYPE = ref [];

# 对空匿名正则使用 ref
my $REGEX_TYPE = ref qr//;

# 对空匿名散列使用 ref
my $HASH_TYPE = ref {};

# 对空匿名子程序使用 ref
my $CODE_TYPE = ref sub { };
```

现在我们得到了各种类型引用的名字，这么做不但避免了潜在的拼写错误，还不用花时间记忆类型名字：

```
die '...' unless ref $regex eq $REGEX_TYPE;
die '...' unless ref $array eq $ARRAY_TYPE;
```

我们也可以用这种方法定义 constant 类型常量，或者用 Readonly 模块定义只读属性的类型变量：

```
use constant ARRAY_TYPE => ref [];

use Readonly;
Readonly my $ARRAY_TYPE => ref [];
```

www.TopSage.com

如果需要对子程序参数作更严格复杂的验证，可以试试 Params::Validate 模块。

3. 要点

- 在解引用之前先验证一下引用类型。
- 用原型值比较引用类型。
- 最好将原型值转为常量使用。

条款 60　通过引用创建包含数组的数组

本质上，Perl 没有包含列表的列表，但我们可以通过构建包含其他数组引用的数组来实现。一般我们把这种数据结构叫做**数组的数组**（array of arrays），简写为 AoA。记住，对下标元素使用圆括号还是方括号，必须符合预期的数据结构类型（见条款 61）。

我们可以直接构造数组的数组：

```
# 一个数组，包含指向其他数组的引用
my @a = ( [ 1, 2 ], [ 3, 4 ] );
print $a[1][0]; # gives 3

# 指向一个数组的引用，该数组包含指向其他数组的引用
my $a = [ [ 1, 2 ], [ 3, 4 ] ];
print $a->[1][0]; # 得到 3
```

我们也可以用编程的方式创建数据结构。比如创建一个代表乘法表的矩阵，下面的代码使用了 C 风格的 for 循环语句，逐个遍历所有下标：

```
my $max = 5;
my $matrix;
for ( my $i = 1 ; $i < $max ; $i++ ) {
  for ( my $j = 0 ; $j < $max ; $j++ ) {
    $matrix->[$i][$j] = $i * $j;
  }
}
```

要漂亮地打印出一个多维数组并不难，绝大部分的工作在于调整输出格式，而访问这种数据结构取得数据实际上非常简单：

```
my $format = ' %2d' x @{$matrix};

printf " i/j $format\n", 0 .. $max;
for my $i ( 0 .. $max - 1 ) {
  printf "%2d: $format\n", $i, @{ $matrix->[$i] };
  }
```

下面是已经格式化的漂亮的乘法表：

```
i/j  0  1  2  3  4
0:   0  0  0  0  0
1:   0  1  2  3  4
2:   0  2  4  6  8
3:   0  3  6  9 12
4:   0  4  8 12 16
```

www.TopSage.com

大家网 TopSage.com

如果想得到某个特定元素，直接用该元素相应的下标索引就好了：

```
my $two_squared = $matrix->[2][2];
```

对第一个下标需要注意，我们得在 $matrix 后使用 -> 符号，因为它是一个引用。而在第一个下标索引之后，我们就可以省略 -> 符号了，因为两个相邻的下标之间已暗藏了引用(见条款 58)。

来看另一个例子，假设需要快速访问文本中的某几行：

```
my @lines;
while (<>) {
  chomp;
  push @lines, [split];
}
```

在用上面的代码创建了相应的数据结构后，我们便可以迅速访问任意一行任意位置上的单词了。比如想要第 7 行的第 3 个单词，只需给出正确的下标就可以了。因为@lines 是个数组（不是引用），所以这里不需要用 -> 符号开头[①]：

```
my $third_on_seventh = $lines[6][2];
```

文件中的第 15 行有多少单词呢？因为这个数组的元素是指向另外一个数组的引用，所以我们可以对这个元素进行解引用，随后赋值给标量变量获得该数组元素个数（见条款 9）：

```
my $count = @{ $lines[14] };
```

那么，哪一行的单词最多呢？我们可以用施瓦茨转换（见条款 22）对数组@lines 的下标进行排序，这么做的同时还避免了创建另外一份数据副本的开销。由于我们最终只需要一个指向包含单词最多那行的下标，所以最后赋值时只需提供一个单个标量变量的列表：

```
my ($most_words) =
  map { $_->[0] }
  sort { $b->[1] <=> $a->[1] }
  map { [ $_, scalar @{ $lines[$_] } ] } 0 .. $#lines;

print "Line $most_words is the longest with ",
  scalar @{ $lines[$most_words] }, " words\n";
```

以上是比较详细的写法。我们也可以用一行较长的列表操作一步完成，正如下面这个施瓦茨转换的变体，有兴趣的读者可以仔细揣摩其中的妙处：

```
use 5.010;

printf "Line %s is the longest with %s words\n",
  map  { @$_ }
  sort { $b->[1] <=> $a->[1] }
  map  { state $l = 0; [ $l++, scalar @$_ ] }
  map  { [split] }
  <>;
```

在 **perldsc**（Perl Data Structures Cookbook，Perl 数据结构大全）文档及 **perllol** 文档中还有许多更加详细的范例可供参考。

① 还记得吗？这里的 $ 仅仅表示取标量值这个动作。——译者注

www.TopSage.com

要点

- 在 Perl 中用数组引用创建包含数组的数组。
- 用数组的数组表示矩阵。
- 使用某个变量之前，要明确它是常规数组还是数组引用。

条款 61　别将匿名数组和列表直接量搞混淆

匿名数组的构造语法 [] 看起来和包围列表直接量外面的那对圆括号很像。表面上来看，它们都是为同一个目的服务的，那就是创建列表。但匿名数组构造器和直接量的列表有很多显著的不同之处。

匿名数组构造器返回的是一个引用，而不是列表。这个匿名数组构造器的目的，就是允许我们直接创建指向数组对象的引用，且无需事先创建一个具名数组：

```
# 如果不用[]，我们可以这样实现：
{ my @arr = 0 .. 9; $aref = \@arr }
print $$aref[4]; # 得到 4

# 或者这样：
my $aref = do { \( my @arr = 0 .. 9 ) };

# 但我们完全可以直接使用[]创建数组引用：
my $aref = [ 0 .. 9 ];
```

虽然我们可以把匿名数组构造器创建的数组引用赋值给某个数组变量，但一般而言这么做并非我们的本意。所以在同时操作数组变量和列表时，或是同时操作标量变量和匿名数组构造器时，得务必小心，不要混淆概念：

```
# 可能原本是想用圆括号的吧？
my @files = [ glob '*.c' ];
print "@files\n"; # 输出类似 ARRAY(0xa4600)的信息

# 另外一个典型的例子——打印出奇怪的结果
# ARRAY(0xa45d0),ARRAY(0xa4654),ARRAY(0xa4558)
my @two_d_array = [ [ 1 .. 3 ], [ 4 .. 6 ], [ 7 .. 9 ] ];
foreach my $row (@two_d_array) {
  print join( ',', @$row ), "\n";
}
```

操作符和函数会创建列表或标量上下文。匿名数组构造器是操作符，而圆括号不是。仅仅用圆括号将某些东西括起来是不会把标量上下文环境转换成列表上下文环境的（见条款 12）。

关于这点，我们可以自己验证：

```
sub arrayish { print "arrayish\n" if wantarray }

my $foo   = arrayish();          # 标量上下文
my $foo   = ( arrayish() );      # 仍然是标量上下文
my $foo   = ( arrayish(), () ); # 还是标量上下文
```

www.TopSage.com

```
my $foo   = [ arrayish() ];      # 现在是列表上下文了
my ($foo) = arrayish();          # 这样也是列表上下文
```

上面列出的只是一小部分可能出现问题的例子，而造成这些问题的原因都一样，无非是本来应该将匿名数组构造器赋值给某个标量变量的，却错误地将列表直接量赋值给了标量变量。另外还有可能出现的问题是，当我们对标量变量解引用时，Perl不管你这个标量变量里的值，它都取出来进行解引用——也许会被它解释成是某个软引用也说不定。而这样得来的结果，显然是毫无意义的。若这里提到的两种情形一起发生，则会形成非常奇怪的行为，调试起来更加费神，比如：

```
# 我敢肯定这里其实本是想用方括号的
my $file_list_ref = ( glob '*.c' );
print "@$file_list_ref\n"; # 什么都没打印出来?
print "$file_list_ref\n";  # 打印出 foo.c 或其他东西
```

联系之前提到的内容，聪明的读者应该可以推敲出这里实际发生的状况了吧：

```
my $aref = ( 1 .. 10 );
print $$aref; # 什么都没打印出来?
print $aref;  # 还是什么都没打印出来?
```

要点

- 用匿名数组构造器 [] 创建引用。
- 匿名数组构造器提供的是列表上下文。
- 将列表赋值给数组，而将匿名数组赋值给标量。

条款62 通过匿名散列创建C风格的struct结构

人们经常问起“Perl是否有和C一样的真正的数据结构”。好吧，从某种意义上说，它有。我们已经知道Perl里面只有几种为数不多的数据类型：标量、数组、散列、子程序，外加一些奇特的东西及文件句柄。它并没有提供像C或Pascal那样的struct结构体类型的数据。一句话，Perl其实是无结构的。但从另一个角度来看，散列能提供极为相似的效果：

```
$student{'last'}  = 'Smith';
$student{'first'} = 'John';
$student{'bday'}  = '01/08/72';
```

当用到一个散列元素时，只要键名是合法的Perl标识符，我们就可以省略它两边的引号：

```
$student{last} = 'Smith';
```

瞧，这看起来不正像是结构体的一个元素吗?

有人看到这个的第一反应可能会是：“啊！这不就是通过某个字串在结构体中查找相应的成员吗！这也太没效率了！真正的结构体应该使用某种由编译器计算好的数字偏移量来定位其中的成员。”。事实上，这种问题Perl已经考虑到了，我们无需为此担心。因为Perl是一门解释型语言，所以访问变量、数组或者散列元素时，总会相对慢一些。其实在散列中查找某个元素所花费时间相对其他任务来说真的是微不足道。

www.TopSage.com

像这样的“结构体”还可以直接传递给子程序处理：

```
sub student_name {
  my %student = @_;
  return "$student{first} $student{last}";
}

print student_name(%student);
```

现在看起来好像不错，但实际效率并不高。当我们把散列当作参数传递时，子程序会在内部将整个散列展开为列表复制过来。另外，上面这样的写法在语法上也有一定的局限性，比如无法同时传递两个散列。因为 Perl 在调用子程序时会先把所有传入的散列解开成一个列表，随着子程序内部两个散列最终合并成一个列表，也就无从区分了。像下面这样的写法，传进来的所有键值都会被第一个散列 %roomie1 全数吞下：

```
sub roommates {
  my ( %roomie1, %roomie2 ) = @_;
  ...
}
```

所以，单纯传递散列并不完美，但如果使用散列引用，问题就迎刃而解了。实践中，我们经常直接用匿名散列构造数据结构：

```
my $student = {
  last  => 'Smith',
  first => 'John',
  bday  => '01/08/72'
};
```

我们也可以事先创建一个空结构，然后一点一点往里填充。用箭头语法访问“结构体”中的成员，看起来就和 C 或 C++ 中一样：

```
$student          = {};
$student->{last}  = 'Smith';
$student->{first} = 'John';
$student->{bday}  = '01/08/72';
```

此时我们操作的就是标量而不是散列了，将它们传递给子程序也会更加高效，并且一次传递多个散列也没问题：

```
sub roommates {
  my ( $roomie1, $roomie2 ) = @_;
  ...
}

roommates( $student1, $student2 );
```

其实这种技术也是 Perl 面向对象编程时用来构造对象变量的基础[①]。

① 散列结构方便保存对象数据，虽然数组也可以 bless 为某个类下面的对象实例，但用散列更自然，所以多数对象都是在散列结构的基础上构建的。——译者注

www.TopSage.com

要点

- 使用散列可以模拟C的结构体struct。
- 用散列引用代替散列传递给子程序，以避免使用额外的数据副本。
- 给子程序传递散列引用，以保持原始散列的独立性。

条款63　小心循环数据结构

Perl使用引用计数的方式管理内存。每当某个对象被命名或有新的引用指向它时，Perl就会增加这个对象的引用计数。而一旦某个对象的名字被销毁或失去指向它的引用时，Perl就会减少它的引用计数。直到引用计数变为零，Perl会自动删除这个对象并回收它所占用的存储空间。

当不同对象之间相互保持引用计数，或者对象内在保持自身引用的时，就会形成循环引用计数，于是通过引用计数自动释放占用资源的机制就会失效。请看下面这个例子：

```
package Circular;

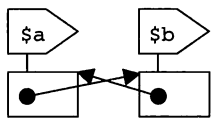
sub new {
  my $class = shift;
  return bless { name => shift }, $class;
}

sub DESTROY {
  my $self = shift;
  print "$self->{name}: nuked\n";
}

package main;
{
  my $a = Circular->new('a');
  my $b = Circular->new('b');
  $a->{next} = $b;
  $b->{next} = $a;
}

print "the end\n";
```

main包中的代码块创建了两个属于Circular类的对象，这两个对象各自都包含了指向对方的引用。在代码块运行结束之前，这种情形看起来类似下图：



每个对象的引用计数都是2：对象命名的时候算一个，然后在另一个对象中的引用又算一个。当代码块退出时，词法变量$a和$b的作用域也就失效了，此时情况看起来像：



www.TopSage.com

现在我们已经无法再取得这些对象了，因为既没有名字，也没有额外指向它们的引用，就好像断了线的风筝，我们知道它们就在内存里，却没法对它们操作。很遗憾，目前 Perl 对这种情况无能为力，因为两个对象各自都还有一个引用计数，它们会一直耗着，直到整个程序退出为止。

不过请放心，这些对象最终还是会被 Perl 销毁掉的。在线程执行的最后一刻，Perl 会用“标记—清除”（mark-sweep）型垃圾回收器扫荡一遍。所有由解释器创建的对象，不管是可访问的还是无法访问的，在这一刻都会被销毁。如果我们运行上面的例子，就可以见证最后的这段操作[①]：

```
the end
b: nuked
a: nuked
```

和我们期望的一样，Perl 会按照正常流程执行完程序中最后一条表达式后，自动销毁运行期间创建的所有对象。

最后这个过程很重要，因为我们可能会把 Perl 当作嵌入式语言来用。如果我们在同一个进程中反复不间断地运行上面这样的代码，而又没有相应机制来销毁那个线程中创建的所有对象，就会发生内存溢出问题。

如果程序长时间运行不会中止，那一旦发生上面的问题，就没办法清理[②]，但我们可以适当地利用蛮力手段防止这种情况的发生，比如采用显式切断循环引用的技术。就之前的例子而言，一个可用的解决方案是：

```
package main;
{
  my $a = Circular->new('a');
  my $b = Circular->new('b');
  $a->{next} = $b;
  $b->{next} = $a;
  $head      = $a;
}
undef $head->{next};
undef $head;
```

这里我们借助变量 $head，构造了一个链表入口，由于这个循环数据结构只包含一个环，只要销毁这个单链表，便可以回收其中所有对象所占用的内存。如果觉得这个方法还不够彻底，我们可以试试亲自销毁每个对象：

```
while ($head) {
  my $next = $head->{next};
  undef $head->{next};
  $head = $next;
}
undef $head;
print "the end\n";
```

① 对象销毁前，会自动运行名为 DESTROY 的对象方法，我们籍此打印该信息。

② 因为从 Perl 的角度来看，它根本就无从分辨，自然无法清理。——译者注

www.TopSage.com

在上面的例子中，我们遍历了整个数据结构并显式地销毁了所有可能引起问题的引用。由于我们销毁了所有想删除的对象的引用，所以这些对象的引用计数就都变成了零。想要显式销毁某个对象却不影响其引用计数，这在 Perl 中是不可能的，否则一定会造成严重错误，甚至程序崩溃。

另一种方法类似于“标记—清除”型回收器，但需分两步走。首先获取想要销毁的引用清单或“目录”：

```
my $ptr = $head;
do {
  push @refs, \$head->{next};
  $head = $head->{next};
} while ( $ptr != $head );
$ptr = $head = undef;
```

这个循环会遍历存在自引用情况的数据结构，然后收集一张列表，包含所有需要销毁的引用。下一步就是遍历这个列表并逐个销毁：

```
foreach (@refs) {
  print "preemptive strike on $$_\n";
  undef $$_;
}
```

这种分两步走的方法对于示例这样简单的情况来说，似乎有些大材小用，但对于具有多个循环的复杂数据结构（比如用以显示图表的数据结构）来说，它可能是唯一的解决方法。

要点

- Perl 靠引用计数来管理内存。
- 避免写出 Perl 没法进行回收的循环引用。
- 如果我们创建了循环引用，就得自己销毁这些循环引用。

条款 64　用 map 和 grep 操作复杂数据结构

我们常常需要从某个多维数组或散列中提取一段数据切片，特别是符合一定条件的数据切片。有时也会需要将这类小的数据集合为大的二维数组，或者进一步组合成三维数组。对于这类工作，Perl 的 map 和 grep 操作符无疑是最完美的选择。

1. 用 map 切片

让我们从读取三维坐标数据文件的程序开始，假设该文件包含如下数据：

```
# 点坐标数据
1 2 3
4 5 6
9 8 7
```

下面的程序将文件中读取到的三维坐标值载入内存。数据文件中每一行表示一个坐标点，用空格分隔 x、y、z 轴方向的取值：

```
open my ($points), '<', 'points'
  or die "couldn't read points data: $!\n";
```

www.TopSage.com

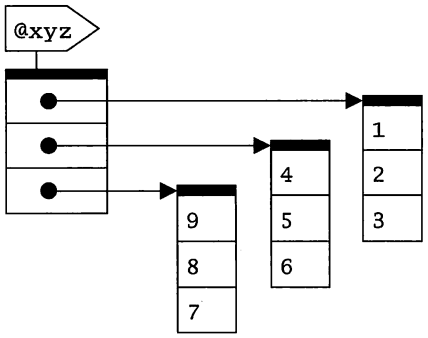
```
while (<$points>) {
  next if /^\s*#.*$/; # 跳过注释
  push @xyz, [split];
}

foreach my $pt (@xyz) {
  print "point ", $i++,
    ": x = $pt->[0], y = $pt->[1], ",
    "z = $pt->[2]\n";
}
```

运行后输出：

```
point 1: x = 1, y = 2, z = 3
point 2: x = 4, y = 5, z = 6
point 3: x = 9, y = 8, z = 7
```

下面是这些点读入内存后的数据结构示意图：



现在，假设我们只需要取得每个点的 x 轴值（也就是数组中 0 号元素），正如上面这张 PEGS 示意图指示的那样，我们可以写一个循环，配合使用明确的索引下标取出数据，比如用 C 风格的 for 循环：

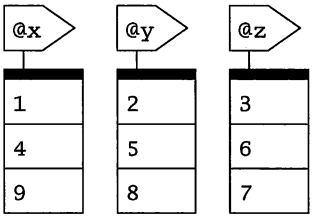
```
for ( $i = 0 ; $i < @xyz ; $i++ ) {
  push @x, $xyz[$i][0];
}
```

不过，改用 map 的话就最自然不过了：

```
my @x = map { $_->[0] } @xyz;
```

2. 用 map 嵌套数据

如果我们有三个平行数组 @x、@y 和 @z，它们分别包含每个点在不同方向上的向量值：



www.TopSage.com

现在，像之前说过的那样，我们希望把这些数据组合成一个三维的数据结构。我们仍然可以用普通循环结构实现，把@x、@y 和@z 中对应的数据逐个拼接成@xyz，但这种方式既慢且麻烦：

```
for ( $i = 0 ; $i < @x ; $i++ ) {
  $xyz[$i][0] = $x[$i];
  $xyz[$i][1] = $y[$i];
  $xyz[$i][2] = $z[$i];
}
```

而改用 map 的话，代码则非常简洁优雅。直接在 map 里面用 [] 构造内层数据结构便是了：

```
my @xyz = map { [ $x[$_], $y[$_], $z[$_] ] } 0 .. $#x;
```

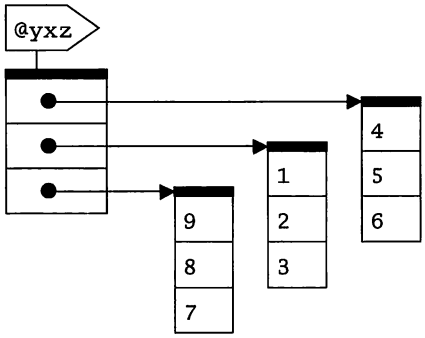
毫无疑问，利用这种方式，我们还可以推测出许多不同的切片和嵌套。比如交换 x（第 0 个元素）和 y（第 1 个元素）坐标的位置：

```
my @yxz = map { [ $_->[1], $_->[0], $_[2] ] } @xyz;
```

同样，利用切片还可以对数组元素进行重新排列，而且这样写看起来又直观又舒服：

```
my @yxz = map {
  [ @$_[ 1, 0, 2 ] ]
} @xyz;
```

此时的数据和使用单独数组时比较，有很大不同：



或者，我们还可以根据这些坐标值衍生出到原点距离的列表：

```
my @mag = map {
  sqrt( $_->[0] * $_->[0] +
      $_->[1] * $_->[1] +
      $_->[2] * $_->[2] )
} @xyz;
```

而下面的这段施瓦茨变换（见条款 22）也是利用 map 实现同时提取切片和嵌套数据的操作：

```
my @sorted_by_mtime =
  map { $_->[0] }              # 提取切片
  sort { $a->[1] <=> $b->[1] }
  map { [ $_, -M $_ ] }        # 嵌套数据
  @files;
```

www.TopSage.com

3. 用 grep 进行选择

假设我们需要对 @xyz 中的点进行过滤，让它只包含 y 坐标大于 x 坐标的那些点，我们仍可以用普通的循环结构来做（你也猜到我们会这么说了吧？）：

```
foreach $pt (@xyz) {
  if ( $pt->[1] > $pt->[0] ) {
    push @y_gt_x, $pt;
  }
}
```

但在这里，把任务交给 grep 就再完美不过了：

```
my @y_gt_x = grep { $_->[1] > $_->[0] } @xyz;
```

当然，我们也可以把 map 和 grep 结合起来用。比如为了收集 y 大于 x 坐标点的 x 坐标值，可以这样写：

```
my @x = map { $_->[0] }
  grep { $_->[1] > $_->[0] } @xyz;

my @x = map { $_->[0] > $_->[1] ? ( $_->[0] ) : () } @xyz;
```

4. 要点

- 用 map 可以将一个数据结构中的元素转换到另一个新的数据结构。
- 用 grep 可以从复杂数据结构中选取特定元素。
- 将 map 和 grep 结合使用可以完成复杂操作。

www.TopSage.com

大家网
TopSage.com

第7章 CPAN

CPAN是所有Perl程序及其相关文件的储藏库，收集了大量perl源代码、Perl程序库和模块以及Perl应用程序。有种说法，Perl程序员的能力也是以他能否有效使用CPAN为判断标准的。截至本书完稿时，CPAN的内容已经超过6GB，来自7758位作者的16835个模块包含其中。

1993年，Tim Bunce、Jarkko Hietaniemi和Andreas König组建了**perl-packrats**邮件列表。Larry Wall那时正忙着开发Perl 5，这是Perl发展历程中很重要的一个升级版本，它允许大家通过模块对Perl语言进行扩展。随着Perl变得越来越流行，人们开始创建非常实用的程序库，但此时他们并没有一个官方或统一的方式来发布这些库。对程序库很熟悉的人或许可以在作者网站上找到它们，然后自己下载安装。但如果那个库依赖于其他库，我们就不得不再去找它依赖的那些库，如此再三，直到把所有相关的模块全都安装好为止。

1995年，Jarkko建立了一个FTP仓库（ftp://ftp.cpan.org/pub/CPAN），用以收集散落在各处的与Perl相关的东西，这样大家就可以在这个集中的仓库中找到各种资源。

与此同时，Andreas König建立了PAUSE，即Perl作者上传服务器，用于管理各式Perl模块的发布。大家把各自作品上传至PAUSE，然后由它负责创建索引，之后CPAN主站会对PAUSE进行镜像，添加相应的modules/目录和authors/目录。尽管从文件大小和受关注程度来说，authors/目录包含了CPAN的绝大部分内容，但它并不代表CPAN的全部，因为CPAN的目标是成为Perl相关的综合性网络。

另外，有一系列的服务器直接对CPAN主服务器进行镜像。主服务器与PAUSE同步，而开放给公众使用的镜像站点与主服务器也进行同步。这样，一个巨大的仓库就被复制到遍及全球各地的几百台服务器上。因而，一般来说CPAN主站本身并不直接和用户发生关系。

然而，大部分人对CPAN的体验却是来自CPAN的工具链。也许你已经使用过CPAN.pm或者CPANPLUS，对怎么读取PAUSE索引文件，怎样连接至CPAN服务器并帮我们下载和安装所发行的模块，这两个工具驾轻就熟。

经过这么多年，CPAN这个术语已拥有了多重含义。它和Perl语法中的花括号相同，具有多重意义。有些人认为CPAN是个网站，如Graham Barr的CPAN搜索站（http://search.cpan.org/），或是Randy Kobes的另一个搜索站点（http://kobesearch.cpan.org/）。这些网站提供通向CPAN仓库的接口，但更注重于信息的展现，比如提供来自CPAN Testers（http://testers.cpan.org/）的测试信息，来自CPAN Ratings（http://cpanratings.perl.org/）的排行信息，以及来自CPAN RT问题跟

www.TopSage.com

踪系统（http://rt.cpan.org/）的信息等。

条款 65 以非管理员权限安装 CPAN 模块

如果我们能创建文件，就能安装模块。不过，也许你会不停地说着“但是，但是，但是……”，提出这样或那样受限的状况。别管那些，事实是只要我们能创建文件，就可以安装模块。我们甚至可以不需要任何特殊的权限就能安装 perl 本身（见条款 110）。当然这并不是说不用理会某些约束性的权限，这可不是技术层面上能解决的问题。

Perl 模块并没有什么特殊。它们只是文件而已，我们可以告诉 Perl 在哪里找到模块文件。所以在此之前，唯一需要我们考虑的，只是如何把这些模块放到这些地方去。

当然，首先得选择安装这些模块文件的目录。如果只是自己用，大可以把它们放到我们的用户主目录下的 lib 子目录里；当然，选择其他目录也没问题。为行文方便，以下我们假设用户主目录为 /Users/snuffy。

1. 亲自动手

所有人都应该以传统但稍显复杂的方式安装一遍模块，这样就能体会神奇的 CPAN 工具为我们节约了多少宝贵时间。可能安装到一半你就放弃了，但体验一下也不赖。

假设我们已经下载好并解压缩了某个模块的发行版文件包，并且已进入该发行版目录，正准备进行下一步工作。不同的模块可能有不同的构建安装方式，但思路大体类同。

对基于 MakeMaker 的模块发行版，用 INSTALL_BASE 指定安装路径：

```
% perl Makefile.PL INSTALL_BASE=/Users/snuffy
% make test install
```

对基于 Module::Build 的模块发行版，用--install_base 指定安装路径：

```
% perl Build.PL --install_base /Users/snuffy
% ./Build test
% ./Build install
```

无论是哪一种，结果都是在 /Users/snuffy/lib 这个目录里创建一些子目录，而这个目录我们称为基准目录。我们的模块最后可能会放在 /Users/snuffy/lib/perl5 下面，所以随后我们需要把该目录添加到 Perl 的模块搜寻路径中去（见条款 69）。此外，你还会看到 /Users/snuffy/bin 和 /Users/snuffy/man 目录，分别存放相应的工具脚本和文档。

2. 配置 CPAN.pm

而用 CPAN.pm 安装模块的话，只需告诉它我们希望的安装路径。

我们可以把 CPAN.pm 传递给 ExtUtils::MakeMaker 和 Module::Build 的选项都配置一下。通过运行 cpan 程序，我们可以通过交互式接口配置 CPAN.pm。早期版本的 CPAN.pm 需要用户主动提交保存对配置的修改，所以下面最后一行作了显式提交：

```
% cpan
cpan> o conf makepl_arg INSTALL_BASE=/Users/snuffy
```

www.TopSage.com

```
cpan> o conf mbuild_arg --install_base /Users/snuffy
cpan> o conf commit
```

在迫不及待开始安装模块前，请先检查一下当前的配置。再次运行交互式命令，输入配置参数名，看看输出的是否正确：

```
% cpan
cpan[1]> o conf makepl_arg
    makepl_arg          [INSTALL_BASE=/Users/snuffy]
```

实际上这些配置信息是自动写回 CPAN::Config 或 CPAN::MyConfig 模块文件的，所以手工修改这两个文件也能起到相同的效果。

一切设置妥当之后，只需要一条简单的命令就能安装模块了：

```
% cpan Set::CrossProduct IO::Interactive Getopt::Whatever
```

3. 配置 CPANPLUS

如果喜欢用另一个模块发行版安装工具 CPANPLUS，则可以相应地修改配置文件$HOME/.cpanplus/lib/CPANPLUS/Config/User.pm。当然也可以通过它的交互命令工具进行配置，参数取值和使用 CPAN.pm 时一样，比如下面的例子，选择“Setup installer settings”：

```
% cpanp
CPAN Terminal> s reconfigure
=================>      MAIN MENU      <=================

Welcome to the CPANPLUS configuration. Please select which
parts you wish to configure

Defaults are taken from your current configuration.
If you would save now, your settings would be written to:

    CPANPLUS::Config::User

  1> Select Configuration file
  2> Setup CLI Programs
  3> Setup CPANPLUS Home directory
  4> Setup FTP/Email settings
  5> Setup basic preferences
  6> Setup installer settings

Section to configure: [1]: 6
```

此后只需要按照提示一步一步做就可以了。当所有参数都配置好后，再以下面的命令安装模块：

```
% cpanp i XML::Twig
```

4. 使用 local::lib

local::lib 模块会帮我们推测出合适的配置。默认状态下，它会将模块安装目录设置为用户主目录下。利用 CPAN 安装模块时，只需加载 local::lib，它就会自动在用户主目录下建立一个模块目录：

```
% perl -MCPAN -Mlocal::lib \
```

www.TopSage.com

```
-e 'CPAN::install(Net::MAC::Vendor)'
```

此时 CPAN.pm 会将 Net::MAC::Vendor 模块安装在~/perl5/lib/perl5 目录下。不过我们并不需要知道 local::lib 具体将模块目录设在什么地方，因为只要在程序中使用了 local::lib 模块，它就会自动将相应的模块搜索路径追加到@INC 中去：

```
#!/usr/bin/perl

use local::lib;

use Net::MAC::Vendor;
```

如果想知道设置细节，只需在命令行简单加载 local::lib 模块便可以打印当前配置：

```
% perl -Mlocal::lib
export MODULEBUILDRC="/Users/snuffy/perl5/.modulebuildrc"
export PERL_MM_OPT="INSTALL_BASE=/Users/snuffy/perl5"
export PERL5LIB="/Users/snuffy/perl5/lib/perl5:/Users/ ↵
  snuffy/perl5/lib/perl5/darwin-2level:$PERL5LIB"
export PATH="/Users/snuffy/perl5/bin:$PATH"
```

其实 local::lib 还是允许我们显式指定目录的，要了解详细做法，请阅读它的文档。

5. 要点

- 即使没有系统管理员权限也可以安装 Perl 模块。
- 用 lib 或 local::lib 模块告诉 perl 模块在什么地方。
- 配置 cpan 或 cpanp 将模块安装在自定义路径。

条款 66 拥有自己的 CPAN

我们并不需要连接到网络才能使用 CPAN。哪怕坐在飞越大洋的飞机上，或是行进在沙漠中，甚至是无线网络很烂的会议上，我们都可以通过 CPAN 安装模块。

CPAN 仓库目前已经非常庞大了。自 1994 年开始，它的增长速度就没放缓过。在写本书时，完整的 CPAN 已将近 7GB，超过了一张 DVD 的容量。

不过，实际需要从 CPAN 用到的模块并不多，一般我们都是通过相应的 CPAN 工具安装各个模块的最新版本。所以如果把旧版本都剔除的话，整个仓库的大小就降到 1GB 左右了。当然这部分也在不断增长中，这意味着，CPAN 仍在不断持续发展。

1. 建立一个 MiniCPAN

CPAN::Mini 模块提供了 minicpan 程序，它可以将最新版本的模块复制到我们本地电脑上。在命令行调用该程序，给它一个在 http://mirrors.cpan.org/上列出的镜像站点地址即可：

```
% minicpan -l /MiniCPAN -r http://cpan.example.com
```

当然，这里的 /MiniCPAN 并非一定要在本地系统上，我们可以把它镜像到外部设备，比如 U 盘，这样就能随身携带一个 CPAN 了。

www.TopSage.com

自己动手运行 minicpan 是一件很容易的事，不过，要是能时刻保持一个持续更新的MiniCPAN就更好了。不如把这个命令添加到 crontab（或其他能定时运行程序的地方）中定期自动运行。

我们还可以将 minicpan 的配置选项保存在外部配置文件中，只要在你的用户主目录中创建一个.minicpanrc 文件：

```
# ~/.minicpanrc
local: /MiniCPAN
remote: http://cpan.example.com
```

其实我们也没必要把所有最新版的模块都下载下来。如果觉得 minicpan 还不够灵活，可以试试看用 CPAN::Mini 模块自己编写一个工具脚本。比如只关心 MiniCPAN 中一部分内容的话，可以通过 path_filters 或 module_filters 参数进行过滤，也可以直接告诉 CPAN::Mini，以便跳过名字符合匹配条件的那些模块或文件：

```
use CPAN::Mini;

CPAN::Mini->update_mirror(
  remote => "http://cpan.example.com",
  local => "/MiniCPAN",

  # 若能匹配正则表达式，或指定子程序返回真，则跳过不更新
  # 比如类似 B/BD/BDFOY/Mac-iTunes-1.23.tar.gz 这样的路径
  path_filters =>
    [ qr/BDFOY/, sub { $_[0] =~ /JMCADA/ } ],

  # 跳过匹配该正则表达式的模块：
  module_filters => [qr/Acme/i],
);
```

2. 使用自己的 MiniCPAN

现在我们建立了自己的 MiniCPAN，接下来要告诉 CPAN 工具该到什么地方找到它。对CPAN.pm 来说，需要修改 urllist 设置，运行 cpan 命令：

```
% cpan
cpan[1]> o conf urllist push file:///MiniCPAN
Please use 'o conf commit' to make the config permanent!

cpan[2]> o conf commit

cpan[3]> o conf urllist
    urllist
        0 [file:///MiniCPAN]
```

而对于 CPANPLUS，一样启动命令行工具，运行 s reconfigure：

```
% cpanp
CPAN Terminal> s reconfigure
```

从菜单中选择 Select Hosts 选项，接着选择 Custom 选项。

当我们把喜欢的工具都设置好后，它就会从 MiniCPAN 直接下载内容进行安装，而不再连网使用公共仓库。

www.TopSage.com

TopSage.com

3. 注入自己的模块

MiniCPAN 带来的，还有另一个很棒的特性，就是我们可以把自己非公开的模块注入到 MiniCPAN，然后通过标准的 CPAN 工具安装。具体实现可以通过 CPAN::Mini::Inject 模块进行。

首先我们要为 CPAN::Mini::Inject 建立一个配置文件。其格式、内容和之前我们给 CPAN::Mini 创建的配置文件类似，不同的只是存放路径：$HOME/.mcpani/config。以下是配置样本：

```
local: /Users/clara/cpan
remote: ftp://ftp.cpan.org/pub/CPAN
repository: /Users/clara/internal-modules
passive: yes
dirmode: 0755
```

此处我们告诉 CPAN::Mini::Inject，本地仓库存放在 /Users/clara/cpan，公共模块镜像取自 ftp://ftp.cpan.org/，另外将我们自己的私有模块放在 /Users/clara/internal-modules。

现在设好了 CPAN::Mini::Inject 的配置文件，接下来构建一个私有模块，然后放到 MiniCPAN 中去。假设模块已经写好，目前位于模块主目录下，首先要做的就是生成用于发布的打包文件。如果用 ExtUtils::MakeMaker 模块构建的话，运行：

```
% make dist
```

或者用 Module::Build（见条款 79）的话，运行：

```
% ./Build dist
```

这两个命令都会创建一个后缀为.tar.gz 的压缩文件，它包含构成模块发行版的所有文件。如果你愿意，就可以将这个压缩文件直接上传到 CPAN 了（见条款 70）。

如果模块名字是 Foo-Bar，而且当前版本号为 0.01，那么生成的发行版文件名就会是 Foo-Bar-0.01.tar.gz。根据这些信息，我们就可以告诉 CPAN::Mini::Inject 所添加的模块的附属信息。一般我们可以通过 CPAN::Mini::Inject 自带的 mcpani 工具完成这项任务：

```
% mcpani --add module Foo::Bar \
    --authorid JMCADA --modversion 0.01 \
    --file Foo-Bar-0.01.tar.gz
```

接下来，我们需要告诉 mcpani 从公共仓库更新我们的镜像，随后将我们的模块注入，这个操作可以分两步：

```
% mcpani --mirror
% mcpani --inject
```

当然也可以一次搞定：

```
% mcpani --update
```

完成之后，我们就拥有了一个既包含私有模块又包含所有公共模块的 MiniCPAN，以便我们随时从中提取所需模块进行安装。

4. 要点

- 用 CPAN::Mini 维护私人 CPAN 镜像。

- 配置 CPAN 工具，让它指向本地仓库。
- 用 CPAN::Mini::Inject 将私有模块托管至 MiniCPAN。

条款 67　减少公共代码带来的风险

在下载 CPAN 模块前，请勿必考虑清楚！也许我们最初会觉得这事儿轻而易举，用得着瞻前顾后吗？是否存在脆弱的依赖关系？维护者是否靠谱？是否还有没解决的 bug？用这个模块，会带来什么样的结果？

一下子抛出那么多问题，好像我们在阻止你使用 CPAN 模块似的，可你明白，我们本意并非如此。没人能为你做决定，只有你才知道自己当前面临的问题是什么。就好比你现在的收入来源，多半和其他人不同。而提供实时金融数据给交易员，与离线分析 Web 服务器流量这两件事，本身对风险控制的需求就有很大差异。

所有事情都会有折中取舍。使用别人已经写好的模块，确实能让我们节省大量开发时间。而大多数情况下，原作者已对实际问题已经有了深入细致的考虑，提供的实现方案也相当稳健实用。可一旦在你使用时出现问题，你的老板或客户才不会管这些代码的原作者是谁，只会把责任归咎于你。

1. 只是去商店，你并不需要开法拉利

人们常认为处理日期是很简单的问题，但又有多少人了解当年 Dave Rolsky 为了能让 DateTime 支持时区、闰秒以及大多数人甚至闻所未闻的其他细节，而承受了多少痛苦。所以为了避免重蹈覆辙，直接使用 DateTime 也许是不错的选择。

是的，我们说的是*也许*。DateTime 能处理好所有事情，我们确定可以通过它来得到正确结果，可如果仅仅是为了将纪元时间转换成 YYYYMMDD 格式，我们有必要一定用它吗？如果是处理日志文件，类似这样的操作要重复几百万遍呢。我们真的愿意承担那些并不需要的特性带来的额外内存开销吗？也许你愿意，但你是真的*清楚*你这么做意味着什么，还是兀自乱猜？

2. 真的需要升级吗

一个 CPAN 模块的内部往往包含很多东西，我们可能都见过安装某个模块时列出的长长的依赖列表。你有试过手工安装一个较复杂的模块，并在安装过程中自己跟踪不断出现的新依赖关系，然后一一下载相应的文件吗？这活儿很快会变得一团糟。

CPAN 有一个尚未解决的缺陷：PAUSE 只跟踪模块的最新版本，但这些版本号在遍布 CPAN 的不同发行版之间基本上不存在所谓的一致性。当我们使用 CPAN.pm 或者 CPANPLUS 来升级某个模块的时候，它会自动检视这个模块存在的其他依赖关系，并检查我们已经安装了的版本以及 CPAN 上最新的版本。但，我们并不会被问及是否要去升级这些依赖模块，因为这些工具可以悄悄地帮我们做好这一切。不经询问就自动升级很容易打破系统之前拥有的稳定性。

举例来说，在 2.64 版本的 CGI.pm 模块中，Lincoln Stein 把请求参数默认的分隔符从 & 改成了 ;。使用分号是更现代的做法，但很多老代码仍在用 &，因为它是已经发布过的接口的一部分。尽管它们并不需要（甚至都不知道）新版本中的新特性，程序还是会升级到最新的 CGI.pm，问题就出现了。

www.TopSage.com

我们并不是叫大家不要升级，只是想以此说明盲目行动有风险，行动前一定要仔细考虑清楚。当我们知道有潜在风险的时候，可以先在开发机上升级试一试，看看是否有错误出现。

3. 高阶模块

`Test::More` 和 `ExtUtils::Makemaker` 模块的作者是 Michael Schwern，哪怕他在写这些模块时犯一个最微小的错误都可能导致所有人使用 CPAN 都出问题（当然，他本人为了不让类似悲剧发生而经受着巨大的痛苦）。

大部分较现代的模块发行版都依赖于 `Test::More` 处理测试集，如果 `Test::More` 没法正常工作，那其他模块也就没法正常通过测试，而我们的工具链也就无法安装这些测试失败的模块。

在小范围情况下，我们依赖的某些模块可能会导致应用程序出问题。它们可能会带来各种 bug，导致修改接口、改变输出等和以前工作不一致的情况，从而使程序没法正常工作。这种情况的发生是一种必然：要么你以前已碰到过，要么你以后会碰到。所以升级前，要充分考虑这些风险，尽量不要先在生产机上盲目升级。

4. 先尝后买

在冒险给我们的项目使用某些模块之前，可以先做些试验，以此降低风险。另外，Perl 社区也提供了关于某个模块发行版丰富的附加信息。所以我们并不需要盲目以身犯险。

- *查询 CPAN Testers*

CPAN Testers（见条款 97）收集了所有上传到 CPAN 的模块发行版的测试报告，某些模块发行版会包含成百的测试报告，以涵盖不同的操作系统和不同的 Perl 版本，这种情况并不少见。我们可以在 http://testers.cpan.org/ 上检查模块发行版，也可以在 CPAN Search（http://search.cpan.org/）的模块发行版页面上查看其测试结果摘要。

结合依赖关系信息，David Cantrell 依照不同平台和 Perl 版本进行分析，得出大多数 CPAN 模块发行版的安装成功率。在决定升级或试用某个新模块时，我们可以先在 CPANdeps（http://deps.cpantesters.org/）网站查询一下。如果在代表我们所关心的 `perl` 版本和操作系统的那部分没看到绿色方格，我们就得格外小心了。

- *询问 Google*

如果好奇是否还有其他人也用某个模块，我们可以用自己中意的搜索引擎来看看能找到些什么。这事表面看来好像理所应当，但显然很多人都不习惯这么做。我们可以查查是不是有邮件列表。当有人问起关于某个模块的问题时，其他人会热心回答吗？要多久才能得到答复，15 分钟，当天内，抑或是两星期之后？会有多少解答？多少人在踊跃回复？

如果很短时间内就有许多不同的人给出解答，这显然表明了某个模块的活力和流行度。如果解答经常是来自同一人，还显得有点心不在焉，则意味着使用该模块会让你在很长时间内都孤立无援。

- *区分开发和生产环境*

我们不能在生产环境中测试自己的代码，因为不能把当前用户当成是我们的测试对象。首先应该做的是将我们使用了升级过的模块的代码部署到测试服务器上，以便我们可以确认它能否正常工作以及性能是否满足需求。如果是用虚拟机，这些操作甚至都不需要装配新的硬件。

www.TopSage.com

● 升级不要太频繁

真的需要每天都对我们的模块进行升级吗？也许我们确实需要最新版本中包含的修复，但我们每升级，就意味着需要花时间来管理它，哪怕只有一点。我们可能没有注意到，很多升级所依赖的部分经常也需要升级到更新版本，这些间接的升级需求有时候会制造一些麻烦。而频繁地升级也会占用我们正常的工作时间。因此请考虑为所有升级安排统一的时间，比如说以季度为升级周期，这样我们就可以专心于综合集成的工作。

● 使用 DPAN 来管理依赖

有时候使用 CPAN 对我们来说其实有一定风险。如果想要更多可控性，可以考虑自己架设一个 DPAN，或称为（分布式/非中心化的/暗黑的）Perl 存档网络。它是一个私有化的类似 CPAN 的仓库，我们日常的 CPAN 工具链可以用它来替代公共仓库。这样就由我们自己决定什么模块的什么版本可以收入我们的仓库。`MyCPAN::App::DPAN` 模块可以帮助我们完成该任务。

5. 要点

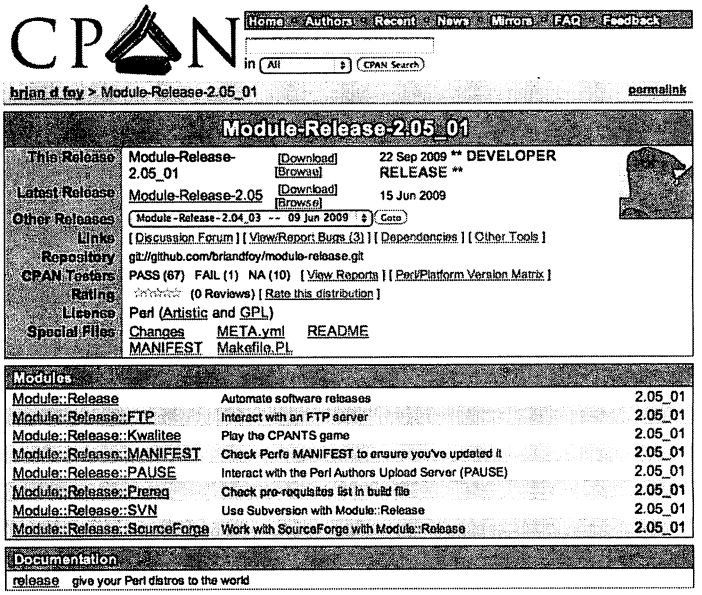
- 使用公共可用的代码是具有一定风险的。
- 在使用模块前先对它调查一下。
- 创建自己的 CPAN 替代品（或“DPAN”），以便掌控确实需要的模块及其版本。

条款 68　安装模块前先行调研

CPAN 就像一座代码的金矿，但它也和现实生活中的金矿一样，里面有很多脏乱差的东西，我们需要舍其糟粕取其精华。幸运的是，Perl 社区中有着丰富的资源，能帮助我们找出这些精华。

1. CPAN 搜索

要想找到关于某个模块或发行版的信息，最佳着手点是 CPAN 搜索（http://search.cpan.org/）。在这个网站上，我们可以找到自己感兴趣的模块，并通过导航进入它的发行版页面。在该页面的顶部，我们就会看到很多由社区提供的资源，如右图所示。作为调研的开始，这确实是个很不错的地方。



2. Bug 跟踪

在 CPAN 搜索的发行版页面上，我们会看到一个能查看和报告 bug 的链接，它会带我们去 CPAN

www.TopSage.com

模块的请求跟踪系统（Request Tracker，简称 RT，网址是 http://rt.cpan.org/）。在这里，我们可以看到该发行版有多少 bug，更重要的是，能了解到有多少 bug 已经被维护者修复了。这能帮助我们对这个模块的质量有一个初步判断，但还有其他一些问题需要我们考虑。

比如说，如果没有 bug 记录，是说明该代码非常稳固，还是没人在用这个模块？相反，如果这儿有很多 bug 记录，是说明这个模块不稳定，还是代表有很多人在用它？如果有 bug 记录，但没回应，是表明这个模块已被放弃，还是该模块的作者并不使用 CPAN 的请求跟踪系统，而是用了别的什么 bug 跟踪系统？

- CPAN Testers

在发行版页面上另一个很重要的链接是指向 CPAN Testers 的（见条款 97）。CPAN Testers 是由专人组成的小组，他们会让发行版在不同平台上运行自动化测试，并将这些测试结果的报告发回供所有人查看。他们提供的信息被概括成三组：成功次数、失败次数和未知状态的次数。

不要被那些失败或是未知状态的记录吓倒。测试者会在多个平台上测试多个版本的 Perl，其中有些 Perl 版本已经很老了。另外也有可能我们所调查的模块，其测试失败的平台不是我们使用的平台，或是它测试的 Perl 是我们老早就淘汰了的版本。我们可以点击详细报告和测试矩阵的链接，以便更好地了解某个发行版真正存在的问题。

- CPAN 评级

对于每个模块发行版，我们还可以看到 5 星制的评级系统，如果有人给它评级，我们将会看到其中一些星变成高亮状态。亮的星越多说明该模块拥有的评价也越高。另外还有一个链接指向大家给这个发行版写的评论。我们可以随意看看这些评论，可能其中会有许多对该发行版的赞扬和批评意见，多少可以作为是否安装它的参考。

评级和评论都是主观的，所以别把它们当作唯一的因素左右我们的最终抉择。不过无论如何，这些信息多少还是有所帮助的。

- 其他考虑

最后一次发布是什么时候，以及发布了多少次？这两个因素可以让我们知道某个模块是否过旧或是已被遗弃。当然，它也可能是已经很稳定而不需要常规升级。

通常都会有一个链接指向该代码的源码控制仓库。我们可以在该仓库中浏览一下，体会一下该发行版的开发情况及活跃程度。说不定我们也能成为它的贡献者！

3. CPANTS 的 Kwalitee

CPAN Testers 还提供另外一个小参考供我们决定什么时候可以用某个发行版。我们可以访问 CPAN 测试服务网站（http://cpants.perl.org/），找到感兴趣的发行版，然后查看它的 kwalitee 值。一种普遍的观点是，我们没法切实判断发行版真正的质量，但可以对某些相近的东西进行考量[①]，即 kwalitee（和 quality 很相近）。

要判断某个模块的质量，Perl 社区认为有一些很重要的因素需要检查，由这些检查项构成的列表就被浓缩为 Kwalitee。比如说，这个发行版是否包含测试和文档，是否遵循了当前的最佳实

① 正所谓，不怕不识货，就怕货比货。——译者注

www.TopSage.com

大家网 TopSage.com

践指南?

当然了，我们也有可能写出运行很完美但 kwalitee 测试结果很糟糕的代码。但事实证明大部分优秀的模块发行版都能通过 kwalitee 中的绝大多数检查，从这点也可以判断出维护者是否注重细节。对于那些 kwalitee 检查失败太多的发行版，我们就得多小心一点。

4. 阅读源代码

Perl 是门开源语言。Perl 代码，尤其是在 CPAN 上的代码，都是文本的，我们想读就能读到的。在下载前我们可以先检视部分代码，以此获知开发者是否对他/她所做的事情了然于胸，当然，或许我们还能从中学到一招半式[①]。

在我们阅读代码的时候，想想看是不是很容易读懂。也许我们并不能一时间理解其中所有特性，但能大体跟上这程序的思路和流程吗？如果需要基于这堆代码进行工作，会不会令自己抓狂？记住，当我们使用了这部分代码，就极有可能需要自己处理它潜在的 bug。

5. 要点

- 在安装和依赖某个模块发行版前先调研一下。
- 用不同的数据源对 CPAN 上的模块发行版进行评判。
- 检查下载的代码，以便在使用前对它多一些了解。

条款69　确保 Perl 能找到我们的模块

那些被 use 指令加载至我们程序中的 Perl 模块，都是文件。那 Perl 是怎么做到只通过模块名就找到这些文件的呢?

1. 包含路径

Perl 会搜索**包含路径**（include path），它是保存在全局变量 @INC[②]中的一个目录列表。

默认的包含路径在编译 Perl 可执行程序时就一同编译进去了。查看包含路径内容的方法有很多，写一段小脚本直接输出是方法之一。我们可以直接在命令行上实现：

```
% perl -e 'print "include is @INC\n"'
```

另一个更简单的可选方式是通过-V 选项查看：

```
% perl -V
Summary of my perl5 (revision 5 version 10 subversion 0) ↵
  configuration:
...blah blah blah...

Characteristics of this binary (from libperl):
  Compile-time options: PERL_DONT_CREATE_GVSV ↵
    PERL_MALLOC_WRAP USE_LARGE_FILES USE_PERLIO
  Built under darwin
```

① 学习的最佳方式就是模仿。——译者注

② @INC 的名字来自于 include path 的前三个字母，很好记。——译者注

www.TopSage.com

```
Compiled at May 8 2009 02:12:43
%ENV:
  PERL5LIB="/usr/local/perls/perl-5.10.0/lib/perl5"
@INC:
  /usr/local/perls/perl-5.10.0/lib/perl5/↵
    darwin-2level
  /usr/local/perls/perl-5.10.0/lib/perl5
  /usr/local/perls/perl-5.10.0/lib/5.10.0/↵
    darwin-2leevel
  /usr/local/perls/perl-5.10.0/lib/5.10.0
  /usr/local/perls/perl-5.10.0/lib/site_perl/5.10.0↵
    /darwin-2level
  /usr/local/perls/perl-5.10.0/lib/site_perl/5.10.0
  .
```

包含路径列出的就是 use 或者 require 指令用来搜索 Perl 模块的目录，而诸如 File::Basename 这样的“嵌套”[①]模块，就位于 File 模块所在的目录树下面。

2. 修改包含路径

如果我们有模块安装在其他地方，而不是在@INC 中设置的目录下面，就需要修改包含路径以便 Perl 能搜索到并加以使用。

这种情况一般不会经常出现。当我们构建和安装模块时，无论是用 CPAN 模块还是手工解压缩构建，对于我们特定安装的 Perl，ExtUtils::MakeMaker 和 Module::Build 模块都会自动搞清楚如何将模块发行版文件放到正确位置。如果不是我们想要的存放位置，可以告诉它正确的路径（见条款 65）。

假设我们有个模块安装在一个很奇怪的地方，比如在/share/perl 这个目录下。

你也许会想到在程序源码中临时修改包含路径：

```
unshift @INC, '/share/perl';    # MyModule.pm 所在的地方

use MyModule;                   # 错误!
```

很遗憾，这个简单的方法工作不了。

use 指令是编译时就处理的，而不是运行时才执行的。当 use Module; 出现在我们的程序中时，它实际上等价于 require 和 import 这两个操作：

```
BEGIN { require 'Module.pm'; Module->import; }
```

Perl 会在编译时而不是运行时执行 BEGIN 区块中的代码。这就意味着在运行时修改 @INC 的内容不会影响 use 指令的执行。

要绕过这个问题，方法之一是将对 @INC 的修改放到它自己的 BEGIN 区块中：

```
BEGIN {
  unshift @INC, '/share/perl';      # MyModule.pm 所在的地方
}

use MyModule;                       # 现在工作正常了!
```

① 即模块名由多个部分构成，以双冒号分隔。——译者注

www.TopSage.com

现在，包含路径在编译时就会设置好，这样 use 就能找到所需的模块了。

尽管这个方法可行，但我们还有更好的方法控制包含路径：使用 lib 编译指令。要将一个或多个目录添加到包含路径之前，只需将它们作为参数提供给 use lib 就好了：

```
use lib '/share/perl';      # 添加目录/share/perl

use lib qw(                 # 添加更多目录
  /xtra/perl
  /xtra/perl5
);

use MyModule;               # 准备好加载了
```

相对于直接修改包含路径，use lib 除了能提升可读性外还有另一个优势，即 lib 编译指令会针对特定的架构平台添加一个相应的自动加载路径（如果该目录存在的话）。当我们使用自动加载的模块时，它就能在性能上为我们带来提升。比如说，如果机器的架构是“sun4-solaris”，那么 use lib '/share/perl'会自动添加/share/perl/sun4-solaris/auto 这个目录，如果这个目录存在的话。

如果 use lib 不合适的话，还有其他一些方法来控制包含路径。我们可以使用-I 命令行选项：

```
% perl -I/share/perl myscript
```

我们也可以在 shebang 行使用-I：

```
#!/usr/local/bin/perl -I/share/perl
```

最后，我们还可以将一个或多个目录名称放到 PERL5LIB 环境变量中。在 UNIX 型的 shell 中，我们可以用冒号来分隔多个目录：

```
% export PERL5LIB=/share/perl:/xtra/perl5
```

而在 Windows 机器中，我们要用分号来分隔多个目录：

```
% set %%PERL5LIB%%=C:/libs/perl;C:/libs/perl5
```

3. 通过 local::lib 使用私有库

很多人的工作环境是受限的，他们无法安装模块至标准 Perl 库目录。尽管我们能将模块安装在任何我们能创建文件的地方，但用户主目录仍是比较好的选择。简单地在命令行上加载 local::lib 会告诉我们它默认在什么地方搜索模块：

```
% perl -Mlocal::lib
export MODULEBUILDRC="/Users/snuffy/perl5/.modulebuildrc"
export PERL_MM_OPT="INSTALL_BASE=/Users/snuffy/perl5"
export PERL5LIB="/Users/snuffy/perl5/lib/perl5:/Users/↵
  snuffy/perl5/lib/perl5/darwin-2level:$PERL5LIB"
export PATH="/Users/snuffy/perl5/bin:$PATH"
```

注意，local::lib 会为我们自动添加这些平台相关的目录。要使用本地库目录运行自己的程序，可以通过命令行 -M 参数进行加载：

```
% perl -Mlocal::lib program.pl
```

www.TopSage.com

我们也可以在程序中使用 local::lib，它会为我们添加正确的目录，相对于 lib 更智能些：

```
use local::lib;
```

我们还可以自动安装模块至 local::lib 指向的目录中去（见条款 65）。

4. 设置相对目录

我们在设置应用程序时可以让库目录贴近程序文件。因为如果当我们把应用程序和库发给他人时，没办法知道别人把它们放在何处，也就没办法预先设置@INC。

在这种情况下，我们可以使用 FindBin 模块来定位当前程序的路径，再构建模块目录：

```
use FindBin;
use lib "$FindBin::Bin/../lib";
```

5. 在编译时设置包含路径

其实在我们编译 Perl 时就可以指定内置的包含路径（见条款 110）。一般来说，我们可以在运行 Configure 脚本时指定安装路径：

```
% ./Configure -des -Dprefix=$HOME/localperl
```

通过改变安装路径，允许我们构建和安装私有的 Perl 或可选的 Perl 副本，以便用于测试或调试[①]。

6. 要点

- 将私有的模块目录添加至@INC。
- 用 local::lib 将模块目录设置到用户主目录下。
- 用 FindBin 添加相对模块路径。

条款 70　为 CPAN 作贡献

表面来看 CPAN 已经有了处理各种各样问题和任务的模块，可它现在的增长速度反而比以往任何时候都快。人们不光是在解决新问题，或是给新软件库创建接口，而更多时候甚至是在发明新的轮子，并且往往这些新轮子要比旧轮子好很多。我们把自己的代码奉献出来提交给 CPAN 社区的同时，也得到了更多回报。

关于如何为 CPAN 作贡献足以写出一本专著，而事实上已经有相关的书籍面市了：Sam Tregar 所著的 *Writing Perl Modules for CPAN*[②]。我们可以从 Apress 出版社免费下载（http://www.apress.com/book/view/159059018X）。不过其实要入门，只需了解一点基本知识就可以了，接下来我们会做一些大致的介绍。

1. 注册 PAUSE

Perl Authors Upload Server（Perl 作者上传服务器），或简称 PAUSE（http://pause.perl.org/），是 CPAN 背后的重要组成部分。人们将各自的模块和程序上传至 PAUSE，再由它来对这些模块

① 关于多版本 Perl 的共存和切换，请试试看 gugod 的 perlbrew。——译者注

② Sam Tregar，*Writing Perl Modules for CPAN*（New York，NY: Apress，LLC，2002）。

www.TopSage.com

进行索引和存档，最后镜像到 CPAN 各自的主服务器上。

从 http://pause.perl.org/ 网站开始，页面左边的菜单中有个“Request PAUSE Account”链接，点击该链接，填好表格，然后 PAUSE 会将这个请求放到处理队列中给管理员查看。所有账户都由人工检查，大部分账户能在一天内建好。如果我们没有收到账户已激活的通知，可以检查一下 *modules@perl.org* 邮件列表的存档记录（http://www.xray.mpe.mpg.de/mailing-lists/modules/），搜索我们提交请求时所使用的用户名，看看是否有管理员作过任何评论，也许你会看到一封账户已建立的电子邮件。如果确实没有收到那封通知邮件，可以请某些管理员帮你完成首次账户访问。

2. 大胆为之

通过 PAUSE 可以上传我们的模块发行版。但它不会对我们工作的质量或是否有用等方面作出评判。当然它会检查我们所用的模块名字是否和现有的冲突。不过就算有冲突，我们也照样可以上传：只不过 PAUSE 不会对有冲突的模块名字进行索引罢了。

CPAN 允许我们上传任何处于早期阶段的代码、实验性脚本以及未经雕琢的想法。它的格言是“早发布，勤发布”。让人们能尽可能早地访问我们的代码，对我们来说其实并没有什么坏处。

我们越早上传，就能越快得到世界各地的 Perl 用户的帮助。如果给模块一个表示正在开发中的版本号（见条款 97），那么在 PAUSE 没索引该发行版的情况下，我们仍能从 CPAN Testers 那里看到测试结果。

- HTTP 文件上传

登录 PAUSE 网站，选择“Upload a file to CPAN”菜单项，按照表格要求填写提交。

- 匿名 FTP

以匿名用户身份登录 ftp://pause.perl.org/，将我们的文档放在 incoming/ 目录中。完成之后返回 PAUSE 中的“Upload a file to CPAN”菜单项，转到页面底部要求上传。可以看到，我们之前上传的模块会孤零零地安置在 incoming/ 中，直到我们发出上传请求才会被移除。

- 使用命令行工具

`cpan-upload` 工具和 `Module::Release` 中的 `release` 工具可以让我们无需登录网站，只要按几个键就能完成上传[①]。

3. 观察测试报告

我们的文件一上传，CPAN Testers（见条款 97）就会立即注意到。这些分散的志愿者会用他们的自动化测试工具下载文件包，运行测试，然后把结果发送回来，一般几小时内就能完成。报告会包括各种平台上的测试结果，有些是我们手头没有的系统，有些甚至从未听说。此外，我们还可以通过配置 CPAN Testers 精细控制报告内容（https://prefs.cpantesters.org/），这就好像拥有了一个免费的 QA（质量保证）部门一样。

4. 讨论我们的模块

在开发一个新模块之前，最好先问问周围的朋友或同事。也许有一个很棒的模块藏匿在

① 可以试试 `shipit`。——译者注

www.TopSage.com

CPAN 上，完全可以轻快好省地完成我们需要的任务。请参考附录，有些资源或许可以帮我们找到那些能回答我们问题的人。

很多作者会考虑“三天规则”。即认为我们现在的任何好点子在三天后应该还是好点子。当我们在最初为貌似很好的点子激动不已的时候，遵循三天规则能让我们冷静一段时间，避免陷入烦恼不堪的境地。当然，这并不妨碍我们为了乐趣而编程，但如果时间有限，这个原则能帮我们更合理明智地运用时间。

而对于某个新模块，以下这些问题是务必需要明确的：

- *我是在浪费时间吗*

尽管浪费的是我们自己的时间，但如果现有模块马上能让我们感到满意，为什么不用呢？

- *我们的接口合适吗*

在实现模块前，可能已经写好了使用该模块的范例脚本。但由于模块实际上还不存在，所以我们可以修改范例脚本，按照最自然最合理的方式调整接口形式。哪些方法是用来完成常见任务的？接口是否简单易用，清晰易读，和任务中需要用到它的其他模块是否能保持一致？这都是我们需要仔细考虑的。

- *谁能帮助我*

所有工作并不是非得一个人去完成！哪怕我们是为了乐趣编码。其实有时候能和其他 Perl 程序员一起工作，共同实现各自预想的特性，或者相互分享传播，也是一件非常有意思的事。

不要因为得到不好的反馈而过于沮丧。人们更倾向于表达他们的不快，所以这种负面东西有时候看起来比正面的东西还要重要。不要轻视负面反馈，但也没必要事事放在心上（除非这个人是给我们付支票的人，即便如此，我们还是有可能说服他）。很多现在 CPAN 上的中流砥柱都是从这条路开始的。

5. 欢迎加入 CPAN

好了，就是这些了！我想，现在你应该已经成为 CPAN 作者社区中的一员了。

凡事只有经历过之后才会明白，实际没有想象中那么困难，没必要像以前那样畏首畏脚了。更进一步地，我们应该尽力帮助其他人加入 CPAN，一同奉献。

现在，你已经上传了自己的第一个模块发行版，可以开始考虑准备下一个模块了！

6. 要点

- 获取免费的 PAUSE 账号，上传模块到 CPAN。
- 上传模块到 CPAN 应该趁早，并尽量多更新，以便及早得到他人反馈和帮助。
- 在确定模块名称之前，请先到社区咨询看是否合适。

条款 71 了解常用模块

对 Perl 程序员来说，有很多很多不同的模块可以拿来就用。各种常见编程任务都已封装成了 CPAN 上免费自由分享的模块，包括电子邮件、网络新闻、FTP 及万维网工具，图形和字符用户界面工具，Perl 语言扩展，数据库、文本处理、数学运算以及图像处理工具，等等。这里列出的

只是冰山一角，实际远不止这么一点。

我们没法告诉你该用什么模块，因为我们对你所做的事情并不了解。下面是一份当前流行模块的名单，并不是很长。如果你喜欢的模块不在该名单中，可以发给我们一个备忘，也许我们会在本书下一版中加上它。

在模块 **App::Ack** 中我们可以找到 `ack` 这个用以替代命令行 `grep` 的工具。而在项目网站 http:// betterthangrep.com/ 中有更多信息可供查询。

最流行的 Perl Web 框架非 **Catalyst** 莫属。如果想要新建一个大型 Web 项目，可以考虑使用它。并且它还有一个活跃的社区，我们随时都可以去那里寻求帮助。想要更多信息，可前往 http://www.catalystframework.org/。

在 CGI 方面，**CGI::Simple** 作为老牌的 `CGI` 模块的非正式替代品，显得更轻便，大部分原因是它去掉了 `CGI` 里生成 HTML 的那部分代码。

如果我们的工作需要更精确的日期和时间，想获得闰秒、时区和时间段等各种详细信息，**DateTime** 就是我们所需要的。

DBI 模块是 Perl 的抽象数据库接口，可以兼容 CSV、Sqlite、PostgreSQL、Oracle、ODBC 等流行（或不流行）的数据库服务器。

对那些不喜欢直接用 SQL 的人或更倾向于在运行时自动生成跨数据库的 SQL 语句的人来说，可以选用 **DBIx::Class** 这个非常流行的 ORM[①]模块。

处理 E-mail 可能会很复杂，但是 Ricardo Signes 和其他许多人花了相当多的时间在他的 **E-mail** 系列模块上，而让电子邮件的处理变得更为简单和准确。我们可以用模块来发送邮件、解析邮件、处理 MIME[②]等很多事情。

需要将来自 shell 脚本的环境变量导入我们当前执行的 Perl 程序？只要一条简单的 `use` 表达式，通过 **Env::Sourced** 模块就能完成这个任务。

通过 **Getopt** 系列模块解析命令行选项是件非常容易的事。**Getopt::Long** 既能处理长名称参数，也能处理短名称参数。它现在已经成了许多 Perl 命令行工具的标准配置。

为了兼容各式网页浏览器，网页 HTML 文档的结构常常过于松散。而有着较强包容性的 **HTML::Parser** 解析器，则可以担起解析现实世界中 HTML 文档的任务。而 **HTML::TreeBuilder** 则在高级特性的处理上提供了许多有用的功能。

Image::Magick 是一个很强大的用来创建和编辑图像的软件套件，它是流行的同名软件库的 Perl 接口。

有很多模块可以处理 JSON[③]。而 **JSON::Any** 模块提供了一个通用接口，用以处理各式 JSON 数据的解析和转换。

Log::Log4perl 是 Perl 版本的 Java log4j 日志库。它可以处理各式日志消息相关的细节，包括选择记录哪些消息以及将消息发往何处，等等。有了这个模块，打开和关闭日志消息的记录级别

① ORM，即 Object-Relational Mapper，对象-关系映射，是一种将数据库中的数据按照对象的方式表述和操作的技术。目的是隐藏数据库操作细节，引入更高级的操控行为。——译者注

② MIME，Multipurpose Internet Mail Extensions，即多功能网络邮件扩展协议。——译者注

③ Java Script Object Notation，即 Javascript 的对象表示法，简称 JSON，是一种流行的数据交换格式。——译者注

www.TopSage.com

及流向都会变得轻松简单。

LWP（`libwww-perl`）是用来处理 Web 编程的一个很强大的模块集。**LWP::Simple** 可以胜任绝大多数常见任务而不会让我们犯晕。

如果要抓取 Web 页面或测试 Web 页面的交互行为，可以了解一下 **WWW::Mechanize** 模块，它是基于 **LWP** 模块构建的，该模块以用户使用行为作为界面提供 Web 页面访问的模拟，所以用来作测试或跟踪一系列超链接作检查等，都极为简单而直观。

如果我们需要做大量数字或科学计算，那么 Perl Data Language 即 **PDL** 就是为我们准备的。它是被优化过的一种 Perl 惯用技法，能更快速地以自然语法进行数学处理。

Moose 是 Perl 的下一代面向对象系统。我们倒是希望能多写一点关于它的介绍，但最好还是专门出一本书，详细讲解所有相关的概念、实现和应用。如果你要开始一个新项目，不妨考虑用它作为面向对象架构的基础模型。要了解更多信息，请前往 http://www.iinteractive.com/moose/。

POE 是个多任务网络化的事件处理框架，其内部实现了一个事件循环管理机制，所以我们常用它来处理各式有关事件驱动的编程应用。

我们猜想将来某些历史学家回过头来评价现时的软件，八成会认为微软的 Excel 是最为有用的软件之一。所以 Perl 也毫不例外地支持对 Excel 文件的读写，只要使用现成的 **Spreadsheet::ParseExcel** 或 **Spreadsheet::WriteExcel** 模块就成了。

Template::Toolkit 是用 Perl 驱动的强大的文档模板引擎，能和 **Catalyst** 这样的 Web 框架无缝集成。更多信息请查看 http://template-toolkit.org/。另外，你也许会更中意 **Mason** 或者 **Text::Template** 模板引擎。但不管你选择哪个，通过模板系统我们可以将数据处理和数据表现这两个逻辑的代码分离开来，而且阅读和维护都很容易。

解析通过分隔符隔开的数据貌似非常简单，但当考虑到内嵌分隔符、转义字符以及引号时，情况就不那么简单了。**Text::CSV_XS** 可以搞定常见的分隔型数据的解析工作，而且还很快（见条款 115）。这个模块很重要，我们稍后会在专门的条款中详加讨论。

在 Perl 中解析 XML 结构的文档有很多方法。**XML::Twig** 是个比较成熟的系统，它可以一次解析整个文档，也可以接受流式文档逐项解析。此外，它还支持 XPath 方式定位文档节点等许多实用功能。想要了解更多，可以查看 http://xmltwig.com/。

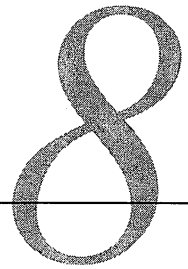
XML::Compile 用 XML Schema 来做 XML 和 Perl 之间的相互转换。该模块可以处理所有基于 XML 实现的数据交换协议，包括 SOAP 和 WSDL。

YAML 是在很多开发者中流行的基于空白字符的表示数据结构的格式。虽然对我们来说它透着一股 Python 的气息[①]，但它仍然是个用起来非常方便的格式化工具，值得了解一下。它正是从 Perl 社区发源的，所以肯定不会糟到哪儿去吧。

① 因为 YAML 严格使用缩进空格表示数据层级，这一点和 Python 用缩进空格表示代码区块类似。——译者注

www.TopSage.com

第8章 Unicode



世界原先是如此简单——所有字符都可以用7比特表示，不必担心特殊字符，也不用考虑各种语言的字符集。那时候，字符串就是字节序列，而每个字节都朴素地代表着一个字符。于是人们渐渐习惯于将字节同字符看作一回事，这种坏习惯一直到现在还都混淆着人们的概念，编程领域也是如此。我们要纠正过来，把表示文字的字节（byte）叫做**位组**（octet）。

现在我们知道语言文字中的字符和字节并非同等概念。要表示各种符号文字或是用以构成新字符的已有字符片段，应该使用 Unicode Character Set（有时也简称为 UCS）。这本书无法涵盖 Unicode 各项细节，不过作为入门，我们至少应该了解一些基本概念。

Unicode 以不同的**代码值**（code point）表示各个字符。每个代码值其实就是一个数字，作为字符的唯一编号。正如 Unicode 学术协会[①]所说：

> 不论是什么平台，
> 不论是什么程序，
> 不论是什么语言，
> Unicode 给每个字符提供了一个唯一的数字。[②]

比如字母 k 对应的数字是 U+006B，这就是它的代码值。

现在不要考虑底层是如何存储这些数据的（将来也没必要），代码值所使用的数字一个接一个形成序列。Unicode 通过定义各种**编码方式**（encoding），完成代码值和位组之间的相互转换。至于位组的表示细节，如果可以就不要细究，相信 Perl，它会帮我们搞定。

在 Perl 的世界里，字符串不再是简单的一种概念，我们有**字符字串**（character string）用以表示字符文字序列，还有我们熟知的原始**二进制字串**（binary string）用以组合成为位组。两种情况 Perl 都可以应付，不过有时我们需要小心，明确 Perl 用正确方式处理不同类型的字串。我们的任务总是同实际使用的文字打交道，所以其他细枝末节就交给 Perl 来做好了。

不仅如此，单是字符字串就有许多种，因为要表示不同语言的文字，需要用到不同的**字符集**（character sets），像 ISO-8859（拉丁字符）、CP-1251（西里尔字母）或是 UCS。最终，Perl 会把

① Unicode 学术协会，即 Unicode Consortium，是制定和发展 Unicode 标准的非盈利性组织。——译者注

② 出自 http://www.unicode.org/standard/WhatIsUnicode.html，2010 年 1 月 24 日的版本。——译者注

www.TopSage.com

字符字串按照字符集的约定转换为位组，以计算机能够处理的方式保存。Perl 会把文字字符**编码**（encode）为对应的位组。UTF-8 是 UCS 字符集的一种编码方式，也是 Perl 默认内部使用的编码方式。反过来，将位组转换为文字字符的过程叫做**解码**（decode）。可以用下面的单行命令查看系统中的 perl 支持哪些字符编码：

```
perl -MEncode -le "print for Encode->encodings(':all')"
```

Perl 会尽其所能将编码和解码的过程隐匿到幕后，多数情况下它会按正确的方式处理字符串。如果对字符串统计字数，给出的就是实际文字的字数，其他内建的字符串处理函数也会按照文字的方式处理。

不过有时缺乏线索，Perl 无法猜透实际使用的字符编码是哪一种，比如直接在源代码中输入字符，或是从某个文件中读取来的数据，又或是从网上抓取下来的页面。这种情况就只能靠人工提供信息继而使其能正确进行处理。

在 Perl 5.8 版本之前，它对 Unicode 的支持还是比较弱，许多人都尝试通过外部模块提供更好的支持。而之前的 Perl 5.6 早在 2000 年就已发布，过于古老，我们就不再回顾了。如果你还在用，那就自己想办法看文档吧，或者，还是回到现代用新版吧！

在写本书时，Perl 的 Unicode 支持仍在改进中（在完全结束前还有许多细节要周到处理），不过绝大多数情况下是没问题的，很少会碰到未支持的特性。

最后还有一条注意事项：接下来我们在源代码或字符串等诸如此类的上下文中谈及 Unicode 时，我们说的都是 UTF-8 编码，也就是 Perl 默认使用的字符编码。在使用文字编辑软件、终端程序等工具时，如果相应的编码配置都正确（当然，怎么才算正确你自己说了算），那剩下的就是我们输入的字符是否正确这种小问题了。

条款 72　在源代码中使用 Unicode 字符

现在的 perl 程序允许直接使用 Unicode 字符，可在源代码中径直输入：

```
print 'I need to work on my résumé';
```

这么做可能存在一些问题。首先，程序源代码的实际编码方式到底是什么？编辑器会如何保存？单从屏幕上来看是无法分辨究竟的。它用的是 UTF-8，ISO-8859，还是其他什么编码？最终保存为文件的编码会不会跟编辑器的设定有关？

其次，就算把它保存为 UTF-8 编码的文件，将来别人打开此文件时未必就会按照 UTF-8 来处理。怎么保证别人的编辑器设置的是正确的编码方式呢？就算读取方式是对的，那保存方式呢？

最后，运行该程序的人可能会完全忽视源代码使用的是 Unicode 字符的事实。那么，Perl 又如何才能分辨实际的编码方式呢？

我们可以用 utf8 编译指令告诉 Perl，应将源代码视为 UTF-8 解析（就这么简单，无需更多）。下面的例子使用的是土耳其语：

www.TopSage.com

大家网 TopSage.com

```
use utf8;

print 'Bu iş kârlı'; # “这是有利可图的买卖”
```

我们甚至可以在标识符名上使用 Unicode 字符，一般常见于变量名。那些整天用希腊字母写公式的人就可以用这种直白的方式编写程序：

```
use utf8;
use Math::Trig;

binmode STDOUT, ':utf8';

my $π = 3.14159265;
my $θ = $π / 2;

printf "cos(θ) is %.2f\n", cos( $θ );
printf "sin(θ) is %.2f\n", sin( $θ );

my $α = atan( 1, 1 ) * 180 / $π;

printf "Angle is %.2f\n", $α;
```

但这样的名字不能用于包名或子程序名，因为它们最终都是要映射到相应文件的，所以必须符合文件系统命名规范。像 Foo::Bar 这样的包名会映射为 Foo/Bar.pm 文件，而其中的 AutoSplit 子程序则会映射为 Foo/Bar/auto/some_sub.al 文件。有些非 ASCII 字符可用于子程序名，比如用 π 作为常量子程序，虽然不会报错，但终究属于非正常用法：

```
sub π () { 3.14159265 }

print "π is ", π, "\n";
```

可要是改用雪人字符☃（U+2603）就不行了：

```
sub ☃ () { 'snowman' }
```

这时的报错信息会十分怪异，让人很难马上看明白具体的出错原因：

```
Illegal declaration of anonymous subroutine ...
```

此外，utf8 编译指令是可以在词法作用域范围内起作用的。如果只是一小段代码需要用到 UTF-8 编码，可以在该词法作用域内声明：

```
{
  use utf8;

  print 'Hyvää päivää'; # “早上好”，这是芬兰语
}
```

反过来，如果想要临时取消 UTF-8 编码解析，可以在词法作用域范围内关闭：

```
{
  no utf8;

  print 'Hyvää päivää'; # 现在我们可不能保证什么了!
}
```

utf8 编译指令仅是通知 Perl 使用 UTF-8 编码解析程序源代码。它并不会改变输入或输出的数据流，或是任何其他来源的数据(见条款 73)。它也不会强制我们所用的编辑器或者 IDE 去做些什么特别的事情，也不会对别的什么工具施加魔力。

要点

- 要在源代码中使用 Unicode 字符，就该使用 utf8 编译指令。
- 部分代码如果不需要用 Unicode 字符，可以选择关闭 utf8。
- 标识符，除了包名和子程序名外，都可以使用非 ASCII 的 Unicode 字符。

条款 73　告诉 Perl 该用何种编码方式

如果按以前的方式写代码，在输出或处理 Unicode 字符时，我们会经常看到“Wide character in...,”这样的警告信息。这是因为 Perl 期望得到普通 ASCII 的地方却遇到了超出 ASCII 范围的宽字符。要搞定这个问题很简单，只需告诉相关的文件句柄应期待哪种字符编码。不过就算没有遇见这样的警告，输入或输出的数据也可能因为误会而搞乱，所以最好还是花点力气明确告诉 Perl 该用什么编码。

光是 Perl 自己知道用 Unicode 处理源代码还不够，当把数据送往别处时，最好能保证接收方同样用正确的编码来处理信息。即 Perl 必须对数据进行正确编码，确保该数据的使用者能正确解码。

1. 设置默认的编码方式

可以用 open 编译指令设定默认编码方式。让所有输入输出文件句柄都使用相同的编码方式，可以这样设置 IO 参数：

```
use open IO => ':utf8';
```

如果仅针对输出文件句柄，可以只设置 OUT 参数：

```
use open OUT => ':utf8';
```

自然，也可以仅针对输入文件句柄进行设置：

```
use open IN => ':utf8';
```

另外，我们也可以同时给定不同的设置，分别声明就行：

```
use open IN => ":cp1251", OUT => ":shiftjis";
```

下面示例中的 -C 开关可以用来启用 Unicode 相关的特性，这样不必修改源代码就可以让 Perl 启用指定的特性。如果我们使用该开关时未添加其他参数，则 Perl 会对所有标准文件句柄都使用 UTF-8 编码方式进行处理：

```
perl -C program.pl
```

通过设置环境变量 PERL_UNICODE，我们也可以有选择性地启用 -C 开关。所有相关细节都可以在 perlrun 文档中找到。

www.TopSage.com

2. 为文件句柄设置编码方式

针对特定的文件句柄，我们可以用 binmode 指定编码方式：

```
binmode STDOUT, ':utf8';

binmode STDIN, ':utf8';
```

在打开文件句柄时也可以这么做，一般只需在 encoding(...)中指定：

```
open my ($out_fh), '>:encoding(UTF-8)', $filename or die;

open my ($in_fh), '<:encoding(koi8-r)', $filename or die;
```

如果用 UTF-8，open 命令还可以这样简写：

```
open my ($out_fh), '>:utf8', $filename or die;

open my ($in_fh), '<:utf8', $filename or die;
```

实际上，任何我们安装的 perl 所支持的编码方式都可以通过这种方式指定：

```
open my ($out_fh), '>:encoding(iso-8859-1)', $file
  or die;
```

3. 要点

- 所有文件句柄都应指定数据的编码方式。
- 用 open 编译指令可以设置所有句柄的默认编码方式。
- 可以用 binmode 为不同句柄分别设置不同的编码方式。

条款 74　通过代码值或名字输入 Unicode 字符

除了直接输入字符串，Perl 还提供了多种方式让获得正确的字符作为数据。对于 0x00 到 0xFF 范围的字符，可以直接使用 Perl 的八进制或十六进制数据表示语法：

```
my $CRLF = "\015\012";

my $CRLF = "\x0D\x0A";
```

高于 0xFF 的代码值，可以在\x 后面跟花括号，把代码值写在里面：

```
print "The smiley face is \x{263a}\n";

print "Watch out for the ",
  "Dread Pirate Fenwick! \x{2620}\n";
```

记不住代码值也没关系，可以用 charnames 模块输入字符名字，又好记又好认，在双引号里按照\N{NAME}的形式将名字写入花括号即可：

```
use charnames qw(:full);
my $arabic_alef = "\N{ARABIC LETTER ALEF}";
```

我们还可以到 Unicode 学术学会网站上查阅所有字符名称列表（http://unicode.org/charts/charindex.html），同样的列表 perl 也有一份，否则就没法转换，它被保存在 lib/5.n.m/unicore/

www.TopSage.com

NamesList.txt 文件中，这里的 5.n.m 是我们的 Perl 版本号。

尽管看起来不是很方便，但 chr 和 pack 也可以用来生成 Unicode 字符，只是不那么直观罢了：

```
my $hebrew_alef = chr(0x05d0);
my $georgian_an = pack( "U*", 0x10a0 );
```

以上这些写法都可以在允许变量内插的双引号（及等效的字符引用操作符）中使用，包括在正则表达式匹配和替换模式内。

1. 从代码值到名字的转换

代码值与它所代表的编码并不一致，它只是数字，字面上我们无法看出它到底代表哪个字符，只有转换为字符名称，我们才能有所了解。如果你知道代码值，就可以用 viacode 子程序返回字符名称：

```
use charnames qw(:full);
my $code_name =
  charnames::viacode(0x2620); # SKULL AND CROSSBONES (即骷髅头)
```

或者用 Unicode::CharName 模块，转换之前先得用 hex 把字符串转为十六进制数字：

```
use Unicode::CharName qw(uname);

# MUSICAL SYMBOL G CLEF OTTAVA BASSA (即音乐符号低八度 G 调)
my $code_name = uname( hex('1D120') );
```

2. 从名字到代码值的转换

反过来要从名称转换到代码值，可以用 vianame 子程序，顺便按照十六进制格式输出：

```
use charnames qw(:full);
my $code_point = sprintf '%04X',
  charnames::vianame("THAI CHARACTER KHOMUT");
```

3. 别名

毋庸置疑，这些字符名称写起来很麻烦，需要的时候每次都输入一整串“ARABIC LIGATURE UIGHUR KIRGHIZ YEH WITH HAMZA ABOVE WITH ALEF MAKSURA ISOLATED FORM”吗？这会让人抓狂的，所以别名应运而生：

```
\use charnames ':full', ':alias' => {
  LONGEST => 'ARABIC LIGATURE ...',
  OMG_PIRATES => 'SKULL AND CROSSBONES',
  RQUOTE => 'RIGHT-POINTING DOUBLE ANGLE QUOTATION MARK',
  LQUOTE => 'LEFT-POINTING DOUBLE ANGLE QUOTATION MARK',
};

binmode STDOUT, ':utf8';
print "\N{LQUOTE}Hello Perl!\N{RQUOTE}\n";
```

charnames 默认就设置了一些别名，例如 LINE FEED 和 CARRIAGE RETURN。所以记不住它们的代码值数字的话，就用别名好了：

```
my $CRLF = "\N{CARRIAGE RETURN}\N{LINE FEED}";
```

www.TopSage.com

```
binmode STDOUT, ':utf8';
print "\N{LQUOTE}Hello Perl!\N{RQUOTE}$CRLF";
```

4. 要点

- 从代码值数字到字符名称的转换可以使用 Unicode::CharName 模块。
- 通过字符名称来表示 Unicode 字符，可以使用 charnames 模块。
- 对于名称很长的字符可以相应地设定简短的别名。

条款 75　位组字串转换到字符字串

理想情况下，数据本身应该包含所使用的编码方式，Perl 就可以自动识别正确处理。遗憾的是，现实世界并非如此。随便扔给 Perl 一些字节，它可不知道那是什么编码的字符，只能指望你告诉它了。不止如此，有时候别人声称的编码方式也可能并不是实际正确的编码方式[①]。

所以最好自己试着解码，看看是否能够阅读这些字符，以便确认所用的编码方式。信人不如信己。

1. 设定编码

让我们先来看看一个代表字节序列的字符串，比如，下面的代码正尝试把十六进制数字转换为对应的字符串：

```
my $hex = '4A2761692070617373C3A9206C27C3'
  . 'A974C3A920C3A0205061726973210A';

( my $bytes = $hex ) =~ s/(..)/chr(hex($1))/eg;
```

Perl 没法知道它究竟用的是什么编码。我们可以用调试模块 Devel::Peek 输出 Perl 看到的是什么：

```
use Devel::Peek;

Dump($bytes);
```

得到的信息根本看不出任何有关编码方式的内容（而且我们还没有给出原始文字，连自己都不知道用的是什么编码）：

```
SV = PVMG(0x830ce4) at 0x80eb50
  REFCNT = 1
  FLAGS = (PADMY,SMG,POK,pPOK)
  PV = 0x228bc0 "...\n"\0
  CUR = 30
  LEN = 32
```

在这个例子中，我们如果预先知道这是 UTF-8 编码的字符，就可以用 Encode 模块把它解码为文字字符（记住，从字节到文字字符的转换叫做解码）：

```
use Encode;
```

① 像垃圾邮件，就经常被刻意伪造，使用接近但不同的字符编码声明内容以混淆视听。——译者注

www.TopSage.com

TopSage.com

```
my $utf8_string = decode_utf8($bytes);

Dump($utf8_string);
```

现在 Dump 显示的标志段中出现了 UTF-8 标记：

```
SV = PV(0x801398) at 0x80ea30
  REFCNT = 1
  FLAGS = (PADMY,POK,pPOK,UTF8)
  PV = 0x259a80 "...\n"\0 [UTF8 "...!n"]
  CUR = 30
  LEN = 32
```

实际上，decode_utf8 只是为了方便把几步工作合到一起而成的快捷子程序，真正幕后解码的是 decode 子程序，像这样：

```
use Encode qw(decode);
decode( 'utf8', $bytes );
```

而且 decode 子程序可以作为对象的方法来使用：

```
my $encoding = 'utf8';

my $utf8_string = do {
  my $obj = find_encoding($encoding);
  die qq(encoding "$encoding" not found) unless ref $obj;
  $obj->decode($bytes);
};
```

如果要解码多次，最好使用对象接口，可以省下不少创建编码对象的开销。

解码之后打印输出，也同样需要指定编码（见条款 73）：

```
binmode STDOUT, ':utf8';

print $utf8_string;
```

打印出来的是这句密文：

```
J'ai passé l'été à Paris!
```

若在终端 shell 里运行程序，看到的不是上面的文字，那多半是因为显示方式的设定有问题。我们用的是 iTerm，需要确认将它设置为使用 UTF-8 编码显示，此外还需要一并设置正确的 $LANG 环境变量。

2. 未知编码

对于已知编码的文字要转成其他编码，可以使用 Encode 的 from_to 函数。该函数会直接修改传给它的字符串，而不是返回转换后的结果：

```
use Encode qw(from_to);
from_to( $string, 'iso-8859-1', 'utf8' );
```

可要是手上的字符串交错混杂着 ASCII、Latin-1 和 Unicode 字符该怎么办？不必奇怪，我们的生活本来就是充满各种可能性的，这种混杂的情形实在很正常。所以 Encoding::FixLatin

www.TopSage.com

模块提供了 fix_latin，它会尽力猜测实际的字符编码，然后转换到 UTF-8。不过有时会失败，无法确定的时候只能靠猜，既然是猜，就一定有猜错的时候：

```
use Encoding::FixLatin qw(fix_latin);

my $utf8_string = fix_latin($stew_string);
```

就算有时看起来工作正常，也可能会出现返回的文字失真的情况。因为有些字节序列同时在多个不同的编码方式中都是合法的，所以 fix_latin 只能尽量选可能性高的那个，显然，使用不同编码转换得到的结果肯定不一样。

3. 命令行参数

从命令行读入的参数所使用的编码，由终端 shell 的会话决定，这不归 perl 控制，所以需要小心处理，正确解码。

首先要确定 shell 环境所设定的字符编码。I18N::Langinfo 模块的 langinfo 子程序可以获取各项本地信息，而指定常量 CODESET 可以让它返回对应的本地字符编码名称：

```
use I18N::Langinfo qw(langinfo CODESET);
my $codeset = langinfo(CODESET);
```

得到编码名称后就可以将参数转换至 Perl 内部编码了：

```
use Encode qw(decode);
@ARGV = map { decode $codeset, $_ } @ARGV;
```

4. 字符串编码

只要你乐意，任何给定的字节序列都可以解码，但解码得到的文字未必是可以阅读的正常字符。像是 Unicode 字符里，就有种称为**组合字符**（combining character）的特殊字符，用于给之前字符添加重音或其他标记，所以如果漏掉这个字符，解码出来的就不是原本那个了。

另外我们来看一条芬兰短句，其中用到一个瑞典人的名字。因为是在 Perl 源代码中输入的，所以它是 UTF-8 编码的字符串：

```
use utf8;

# （Åke 是个非常魁梧的家伙）
my $phrase = 'Åke on aika körmy';
```

用 encode 将之转成字节码：

```
use Encode;
my $bytes = encode( 'utf8', $phrase );

print
  join ':', map { sprintf '%02x', ord($_) }
  split //, $bytes;
```

输出的字节编码为：

```
c5:6b:65:20:6f:6e:20:61:69:6b:61:20:6b:f6:72:6d:79
```

前两个字节为 \xc5\x6b，第一个字节在 UTF-8 中表示大写的 A 顶上一个圈的那个字符，第

www.TopSage.com

二个字节表示字符 k。如果交换一下这两个字节的位置，让 decode 尽力解码，结果会失败，同时无法解码的字符被替换为特殊的**替代记号**（substitution character）0xFFFD。

```
my $not_a_with_ring =
  decode( 'utf8', "\x6b\xc5" ); # \x6b\xfffd
```

显然，这不是我们想要的结果，解码得到的已经不是原来的数据。相反的，我们想要知道每次解码是否出现错误。因此可以给 decode 指定第三项参数，在出现非法字节时采取相应的操作。Encode 提供四种常量，代表四种不同的错误处理方式[①]：

- FB_DEFAULT 将非法字节替换为代替记号。
- FB_CROAK 出现错误时立即退出（die）。
- FB_QUIET 立即中止转换，保留完成了一半的字符串。
- FB_WARN 和 FB_QUIET 类似，但会发出警告信息。

因为 decode 会抛出异常，可以把转换代码放到 eval 里面捕获具体的错误信息：

```
my $a_with_ring =
  eval { decode( 'utf8', "\x6b\xc5", FB_CROAK ) }
  or die "Could not decode string: $@";
```

这就可以侦测到错误的编码转换了，如果 eval 报错，就说明解码出问题了。

5. 要点

- 用 Encode 来解码字节序列，并设置字符串的编码。
- 用 Encoding::FixLatin 猜测字节序列所使用的编码。
- 在使用命令行参数前先行解码。

条款 76 Unicode 字符和属性的模式匹配

在正则表达式中匹配 Unicode 字符，比起以往更为灵活，除了原先我们了解的一些字符集合外，我们还可以对 Unicode 字符的**属性**（property）进行匹配。Unicode 字符对自身并不了解。

1. 匹配特定字符

匹配特定字符的方法简单而直接，要么用字符的名字，要么用十六进制的写法，或者如果知道代码值，也可以直接在 \x{} 里面写：

```
if ( $string =~ /\x{263a}/ ) {
  print "I matched a smiley!\n";
}
```

可以直接用 \N{NAME} 语法写在模式匹配中（见条款 74）：

```
use charnames ':full';
```

① 需要使用 use Encode qw(:fallbacks);引入这些常量到当前的名字空间，或者直接在这些名字前加上它所在的包名，如：Encode::FB_QUIET。——译者注

www.TopSage.com

```
if ( $string =~ /\N{A_TILDE}/ ) {
  print "I found an A with a tilde!\n";
}
```

当然我们也可以用长名称、具体指定的字母、使用的大小写及相关的变体标记等：

```
use charnames ':full';

if ( $string =~ /\N{LATIN CAPITAL LETTER A WITH TILDE}/ )
{
  print "I found an A with a tilde!\n";
}
```

假如有段阿拉伯短语，我们想要知道其中是否包含 hamza 字符。Patrick Abi Salloum 为我们提供的这段短语，说的是骆驼穿越沙漠：

```
use charnames ':full';

use utf8;
my $phrase = 'اجتاز الجمل الصحراء';

if ( $phrase =~ m/\N{ARABIC LETTER HAMZA}/ ) {
  print "I matched a HAMZA!\n";
}
```

需要小心处理，如果要匹配的是**字素**（grapheme）（见条款 77），可能会丢失实际的组合字符，而它可是字素构成的一部分。

2. 匹配属性

在正则表达式内部，\p{PROPERTY} 用于匹配字符的属性，而相应的大写\P{PROPERTY}则刚好相反，是在没有该属性时匹配。Unicode 字符可以自我辨识是否全部大写（uppercase）、标题首字母大写（title case）、全部小写（lowercase）或者都不是（可以到 **perlunicode** 或 **perluniprops** 文档[①]中查阅所有属性）。Unicode 里的这种写法相当于 POSIX 里的方括号定义，就像 [[:digit:]]。

要匹配一个字母，可以用 Letter 属性，任何 Unicode 认为是字母的字符都可以被它匹配，而不论是何种语言：

```
my ($letters) = /(\p{Letter}+)/;
```

许多属性都有短小的简名，像 \p{Letter} 就可以用 \p{L} 代替：

```
my ($letters) = /(\p{L}+)/;
```

这里的 Letter 定义的是一个集合，所有字母字符的集合。如果想把活儿做得更精细些，还可以特定匹配其中的一部分，比如只匹配大写字母：

```
my ($uppers) = /(\p{UppercaseLetter}+)/;
```

① 事实上，目前无法通过 **perldoc perluniprops** 查阅该文档，CPAN 上 Perl 5.12 版也没有该文档，不过相关内容已经涵盖在 **perlunicode** 内。另外 ActiveState 版的 Perl 5.12 确实有该文档，地址为 http://docs.activestate.com/activeperl/5.12/lib/pods/perluniprops.html。——译者注

www.TopSage.com

```
my ($uppers) = /(\p{Lu}+)/;
```

类似的，要匹配一个数字，不管它是什么形式的写法，都可以用：

```
my ($number) = /(\p{Number}+)/;
```

能匹配的具有数字意义的字符实在是太多了[①]，多数不是我们需要的。想看看整个清单吗？运行下面的程序：

```
use utf8;
binmode STDOUT, ':utf8';

foreach my $point ( 0 .. 65535 ) {
  my $string = chr $point;
  next unless $string =~ /\p{Number}/;
  print "$point --> $string\n";
}
```

这个例子中也许大部分匹配的结果不是我们所需的，为了更精确地匹配，我们可以用扩展属性指定特定的字符集合：

```
my ($number) = /(\p{ASCIIHexDigit}+)/;
```

有了 Unicode 后，关于空白字符的定义也更宽泛了，除了我们原来常见的那些以外，还有许多别的空白字符形式。下面的例子将所有空白字符都替换为普通的空格符：

```
$string =~ s/\p{Whitespace}/ /g;
```

如果只是想匹配 Unicode 中特定的一段，比如看看是否有希腊语言，可以指定相应的范围：

```
if ( $string =~ /\p{InGreekExtended}/ ) {
  print "I found some greek!\n";
}
```

3. 自己定义属性

属性本身只是一个用来做匹配的 Unicode 代码值的清单。所以自然地，我们也可以定义自己的属性，以简化工作。在任意包名下定义以 `In` 或 `Is` 开头的子程序，返回字符代码值清单即可。使用的时候需要在 `\p{}` 内指定该包的名称和子程序名，像这样：

```
sub MyProperties::IsMyFavoriteLetters {
  return <<"CODE_POINTS";
33
37
62\t6D
CODE_POINTS
}

foreach (qw( 137 Buster XYZ )) {
  if( /\p{MyProperties::IsMyFavoriteLetters}/ ) {
    print "I found my favorite letters in [$_]!\n";
  }
}
```

① 包括各式语言中的数字及不同写法，像中文的“(七)”也算数字，可以排序。——译者注

www.TopSage.com

大家网 TopSage.com

因为实际上我们定义的字母集合就是[37b-m][①]，所以上面程序运行的结果会匹配两个词：

```
I found some of my favorite letters in [137]!
I found some of my favorite letters in [Buster]!
```

定义属性时，可以包含代码值的取值范围，也可以引用已有的属性，包括自己定义的其他属性，及指定是包含这段范围还是排除[②]，这样就可以方便构建自己需要的字符匹配范围。这种做法十分灵活，完整阐述可以参阅 **perlunicode** 文档。

4. 要点

- 可以通过名称来匹配 Unicode 字符。
- 可以按照字符的属性来匹配特定的 Unicode 字符。
- 可以自己定义 Unicode 字符集合作为属性，方便匹配。

条款 77　同字素打交道，而不是字符

所谓**字素**，是 Unicode 中的一个术语（ http://www.unicode.org/glossary/ ），定义为“文字系统的最小有意义单位”，用于表示一个字，不过很多人误以为字素就是字符。程序员在处理 Unicode 的时候会发现，实际上字素有时可以由一个或多个字符组合而成，具体使用多少字符取决于实现方法。Unicode 本身并不具备满足所有组合情况的代码值，但可以按照既定的办法将字符组合起来形成新的字素，比如加注读音记号等。

假设要表示小写字母 a 和变音符（或者 Unicode 里面的分音符（diaeresis））组合而成的字素 ä，在 Unicode 中有两种办法。一种是用预先定义好的字符名称，因为它是常见的字素，所以直接拿来用就行：

```
use charnames ':full';
my $precomposed =
  "\N{LATIN SMALL LETTER A WITH DIAERESIS}"; # 字长为 1
print "Length of precomposed string is "
  . length($precomposed) . "\n";
```

另一种办法是使用两个字符组合，第一个是要修饰的普通字符，第二个是要组合的注音字符：

```
my $composed =
  "\N{LATIN SMALL LETTER A}" . "\N{COMBINING DIAERESIS}";
```

而现在字长为 2，因为用了两个 Unicode 字符表示一个字素：

```
print "Length of composed string is "
  . length($composed) . "\n";
```

不仅如此，更糟糕的是这两个字符并不相等。Perl 是按照字符来逐个比较的，而非字素，所以下面的条件判断结果为假：

① 33 是数字 3 的代码值，37 是数字 7 的代码值，而 62\t6D 是字母 b 到 m 的范围。其中，'\t' 并非印刷错误，指定字符范围时可以使用一个以上的空格或者制表符分隔，所以这里也可以直接用 62　6D 的形式定义。——译者注

② 通过引用属性名称前加上前缀 + 或 - 实现。——译者注

www.TopSage.com

```
unless ( $precomposed eq $composed ) {
  print "The strings are different!\n";
}
```

这个问题绝对可以加到 Unicode 噩梦清单里了，因为从我们再也不能闭着眼睛比较两个字符串是否相等了。从逻辑上来说，这两个字符串表示的意义相同，但内部表达方式不同。下面的子程序可以检验内部的表示方式：

```
sub show_chars {
  my $phrase = shift;

  my $string = '';
  foreach my $char ( split //, $phrase ) {
    my $name = charnames::viacode( ord $char );
    $string .= "$name\n\t";
  }

  return $string;
}
```

现在用 show_chars 打印出来看看：

```
my $precomposed =
  "\N{LATIN SMALL LETTER A WITH DIAERESIS}";
my $composed =
  "\N{LATIN SMALL LETTER A}" . "\N{COMBINING DIAERESIS}";

print "precomposed:\n\t", show_chars($precomposed), "\n";
print "composed:\n\t", show_chars($composed),       "\n";
```

输出结果显示了两者的不同之处：

```
precomposed:
  LATIN SMALL LETTER A WITH DIAERESIS

composed:
  LATIN SMALL LETTER A
  COMBINING DIAERESIS
```

别着急，接下来我们就介绍解决办法。

1. 常化 Unicode 字串

既然这是个问题，就一定有办法解决。为了处理这种字符和字素的不一致，Unicode 提出了一个**分解**(decomposition)的概念。可以用 Unicode::Normalize 模块把预先组合(precomposed)的字素分解为多个字符。该模块可以做各种常化转换，我们在此略过，只需要知道有个 NFD 函数(Normalization Form D function) 能将字素分解为多个字符：

```
use Unicode::Normalize;

my $decomposed = NFD($precomposed);
print "decomposed:\n\t", show_chars($decomposed), "\n";
print "Decomposed and composed are the same!\n"
  if $decomposed eq $composed;
```

www.TopSage.com

现在我们把单个字符形式的字素变为两个字符的版本了，再做比较的话就是相等的两个字串了：

```
decomposed:
  LATIN SMALL LETTER A
  COMBINING DIAERESIS

Decomposed and composed are the same!
```

2. 正则表达式中的字素

另外还有让人眼前一亮的地方：正则表达式可以理解字素。不管其具体组成形式，你可以用 \X 匹配一个单独的字素：

```
my $composed =
  "\N{LATIN SMALL LETTER A}" . "\N{COMBINING DIAERESIS}";
my ($matched) = $composed =~ /(\X)/;
print "matched:\n\t", show_chars($matched), "\n";
```

\X 的匹配结果捕获到$matched 后用 show_chars 函数显示，得到的确实是两个字符表示的一个字素：

```
matched:
  LATIN SMALL LETTER A
  COMBINING DIAERESIS
```

知道吗？这里的\X 只是一个简写，它实际上等效于两个 Unicode 属性的组合，一个是非记号字符（not a mark）出现一次，另一个是记号字符（mark）出现零或多次，即 (?:\P{M}\p{M}*)（见条款 76）。

现在我们可以把字素从字串中提取出来，逐一处理。有两种简单的办法：

```
# 将字素作为分隔符，保留为独立的数组元素
my @graphemes = split /(\X)/, $phrase;

# 从每个字素开始的地方分片，依次存入数组
my @graphemes = split /(?=\X)/, $phrase;
```

拿到这个字素清单后，如果要对字素进行比较还需小心，因为 Perl 只是按字符逐一比较的，这个问题始终存在，和匹配字素完全是两码事。不过，至少现在我们知道手里所拿到的每个数组元素，确确实实就是完整的字素。

3. 要点

- 记住，相同的字形可能有不同的内部表示方式。
- 对于预组合的 Unicode 字素，可以用相应模块分解为多个字符。
- 用\X 匹配一个完整的字素。

条款 78　谨慎处理数据库中的 Unicode

很抱歉，阅读本节也许对你而言帮助有限，因为各种数据库都有各自处理数据的方式，我们

www.TopSage.com

无法一一回答，而且多数问题不是光靠 Perl 就可以解决的。任何接触或存储数据的东西都有可能把事情搞得乱七八糟。

DBI 模块对于 Unicode 的处理是透明的，它根本不在乎要存取的数据是怎样的，它只负责传送数据。而数据库驱动器，也就是特定的 DBD 模块，负责处理所有细节。其中有些已经支持 Unicode 模式：

```
my $dbh = DBI->connect( # MySQL
  'DBI:mysql:test', 'username', 'pass',
  { mysql_enable_utf8 => 1 }
);
my $dbh = DBI->connect( # PostgreSQL
  'DBI:Pg:test', 'username', 'pass',
  { pg_enable_utf8 => 1 }
);
```

不要被迷惑，这只是告诉驱动器将数据以 UTF-8 编码看待，但实际上传给它的可能根本不是 UTF-8 编码的数据。

我们必须保证数据库设置为能正确处理 UTF-8。有些数据库允许在特定字段上设置 Unicode 相关属性，或者整个表格所有字段都使用默认的 UTF-8 编码。不过就算如此，数据库也不会负责校验拿到的数据是否真是 UTF-8。就算给它不符合 UTF-8 编码的位组字串也照单全收，毫不抱怨。当从数据库取回这些数据时，服务器会告诉你这是 UTF-8 字串，可实际上当然不是（见条款 75）。

另外还需要考虑的是字段长度。设置的时候是 `varchar(16)`，不过存储在里面的会是 16 个字符还是 16 个位组？如果要保存的数据是 16 个字素呢（见条款 77）？

我们能给的最佳建议就是按下面的清单进行检查，最好反复阅读并记住这些事项：

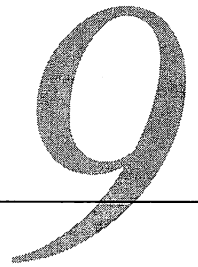
- 如果数据库期待处理的是 Unicode 编码，就要确保每次给它的都是合法的 Unicode 字符串。
- 检查数据库各项设置，确保按照 Unicode 编码存储数据。
- 取回数据时再作检查，以防出错。

要点

- 设置数据库以正确方式处理 Unicode 字段和数据。
- DBI 模块对 Unicode 来说是透明的。
- 在连接数据库时，设置启用 Unicode 支持特性。

www.TopSage.com

第9章 软件分发



虽说任何我们发布的东西都可称为“发行版”（distribution），但论及 Perl 时，人们通常都是指一组 Perl 代码文件及其相关的支持文件，包括了安装该软件用的构建程序的集合。

Perl 的软件分发充满了各种智慧，可以漂亮地自动打包，传送到 CPAN 供人下载，也可以发送邮件报告，或者归到版本控制系统进行管理，任何你喜欢的事情都可以在要发布的软件包内实现。我们在整本书穿插介绍有关软件分发的各式窍门，有些切合主题的已配合着相关章节一并讲了，剩下的部分就集中在本章作介绍。

人们对于软件分发中的问题各持己见，像编程风格中花括号和缩进的使用方式，总是争论不休。是的，完成一件事可以有许多种办法（但要知道，大多数都是错的）。

条款 79 用 Module::Build 构建发行版

最初我们是用 ExtUtils::MakeMaker 构建发行版，它很棒，也足够好。它会从 Makefile.PL 中读取相应配置，然后自动生成 Makefile 文件，只需运行 make 即可执行不同任务，像编译、测试、安装、打包等。不过，因为 Makefile 来自于 Unix 系统，许多它用到的工具在别的操作系统上未必也有，所以多少会有些可移植性方面的问题。而且，为了应付不同系统环境，所生成的 Makefile 也颇为冗长。

如果说有人要安装 Perl 模块，那多半我们可以肯定他已经装好了 Perl 环境，相关的工具基本已经备齐。和 make 不同，Perl 是具有相当的可移植性的，所以为了规避 Makefile 带来的麻烦，我们可以借用 Perl 的力量，由 Module::Build 接替 ExtUtils::MakeMaker 开始新模块的编写。

1. 作为最终用户

改用 Module::Build 后，终端用户并不需要多做什么。即便他们已经习惯于用 ExtUtils::MakeMaker 的方式安装模块：

```
% perl Makefile.PL
% make
% make test
% make install
```

而事实上，使用 Module::Build 模块的话，安装过程也完全类同：

www.TopSage.com

TopSage.com

```
% perl Build.PL
% ./Build
% ./Build test
% ./Build install
```

若是用 cpan 或 cpanp 自动安装模块，用户或许根本就不会注意到这些变化。

2. 作为模块开发者

开始上手用 Module::Build 并不难，如果要新建一个叫做 MyModule 的模块，最简单的 Build.PL 文件只需指定模块名字，以及在哪个文件中能找到相应的版本号即可：

```
use Module::Build;
my $build = Module::Build->new(
  dist_name         => 'MyDist',
  dist_version_from => 'lib/MyModule.pm',
);

$build->create_build_script;
```

我们还可以添加其他相关信息，最常见的是依赖模块清单。不同场合使用的依赖模块，可以分别用 configure_requires，build_requires 或 requires 指定。

Module::Build 可以让我们推荐用户安装 recommends 指定的模块；也可以用 conflicts 提示哪些模块会产生冲突[①]：

```
use Module::Build;
my $build = Module::Build->new(
  dist_name           => 'MyDist',
  dist_version_from   => 'lib/MyModule.pm',
  requires            => { 'Some::Module' => 1.23, },
  recommends          => { 'Some::Module_XS' => 4.56, },
);

$build->create_build_script;
```

甚至还可以更精确一些，直接排除指定版本的模块：

```
requires =>
{ 'Volatile::Module' => '<= 1.2, !1.3, >= 1.4', },
```

借助这种高度灵活的版本限定能力，我们可以编写封装模块以封装其他模块的功能，统一操作接口，规避内部冲突。

3. 定制任务

Module::Build 是以**任务**为中心工作的，这和 make 的工作方式相类似。比如说，运行./Build test，就会执行名为 test 的任务。

有时候我们需要对已建任务做些修改，从 Module::Build 派生子类然后重载相应任务的子程序即可。子程序名为任务名称加上前缀 ACTION_。比如要定制 test 任务，则需重载 ACTION_test。

① 有关这两条命令的具体说明可以参阅 **perldoc Module::Build::API** 文档。——译者注

若是要自己建新任务，取名为 critique，打算使用 Perl::Critic 评测软件代码质量，那么修改 Build.PL 增加一个派生类，在其中定义 ACTION_critique 方法：

```
use Module::Build;
my $class = Module::Build->subclass(
  class => "Module::Build::Custom",
  code => <<'SUBCLASS' );

sub ACTION_critique {
  my $self = shift;
  $self->depends_on("test");
  $self->do_system(qw(perlcritic lib/));
}
SUBCLASS
my $build = $class->new(
  dist_name => 'MyDist',
  dist_version_from => 'lib/MyModule.pm',
  requires => {
    'Some::Module' => 1.23,
  },
  recommends => {
    'Some::Module_XS' => 4.56,
  },
);

$build->create_build_script;
```

4. 要点

- 开始新发行版的开发时，选用 Module::Build 作为构建工具。
- 从 Module::Build 派生子类以创建新任务。
- 在派生子类上调用父类 Module::Build->new 方法返回构建对象，在此对象内定义各项构建任务。

条款 80　不必手工新建软件发行版

每次新建发行版时，都要写一堆模板文件，不胜其烦。本书第 1 版曾介绍使用 h2xs 工具自动生成新模块基础框架（见条款 86）。虽说还能用，但该工具本是为链接 Perl 和 C 代码而设计的。如果单纯用 Perl 写我们自己的软件发行版，现在已经有了更好的工具。

1. 从 Module::Starter 开始

安装了 Module::Starter 模块之后，就可以用命令行工具 module-starter 来创建发行版框架，只需告诉它模块名字及作者信息：

```
% module-starter
  --module=Foo::Bar,Foo::Baz \
  --author="John Doe" \
  --email="me@example.com"
Created starter directories and files
```

www.TopSage.com

module-starter 会自动生成一堆模板文件和目录，基本上所有 Perl 软件发行版都有类似这样的框架：

```
./Foo-Bar/Changes
./Foo-Bar/ignore.txt
./Foo-Bar/lib/Foo/Bar.pm
./Foo-Bar/lib/Foo/Baz.pm
./Foo-Bar/Makefile.PL
./Foo-Bar/MANIFEST
./Foo-Bar/README
./Foo-Bar/t/00-load.t
./Foo-Bar/t/boilerplate.t
./Foo-Bar/t/pod-coverage.t
./Foo-Bar/t/pod.t
```

手工创建可要费不少功夫，没人愿意舍近求远。现在进入该模块 Foo::Bar 的目录 Foo-Bar，已经可以执行构建命令了。

默认会使用传统的 ExtUtils::MakeMaker 作为构建工具，也可以通过指定 --builder 参数改用 Module::Build 或 Module::Install：

```
% module-starter \
  --module=Foo::Bar,Foo::Baz \
  --author="John Doe" \
  --email="me@example.com" \
  --builder=Module::Build
Created starter directories and files
```

如果每次都选用一样的参数，可以在 ~/.module-starter/config 文件中配置默认取值：

```
author: John Doe
email: me@example.com
license: artistic
builder: Module::Install
```

以后只需在--module 选项给出新模块名称即可[①]：

```
% module-starter --module=Foo::Bar,Foo::Baz
```

2. Module::Starter 的插件

在构建完模块框架之后，随着工作展开，可能需要添加一些新的模块文件。默认的 Module::Starter 没办法做，但它的插件 Module::Starter::Smart 可以。

要激活使用插件模块，需在安装后将模块名写入配置文件 ~/.module-starter/config，这样程序启动时，Module::Starter 会先完成基本设置，然后加载插件 Module::Starter::Smart 的功能：

```
plugins: Module::Starter::Simple Module::Starter::Smart
```

现在到模块所在目录，再次运行 module-starter 添加新模块文件 Foo::Buzz：

① 这里给出了两个模块名，实际模块名称由第一个确定，同时在 lib/Foo 下面会生成第二个名称指定的模块。——译者注

www.TopSage.com

```
% module-starter --module=Foo::Buzz --dist=Foo-Bar
Created starter directories and files
```

打开 Foo-Bar/lib/Foo/ 目录会看到新增了 Buzz.pm 文件，而与此同时，文件清单 MANIFEST 中也新增了一行记录，表明 lib/Foo/Buzz.pm 将随同发布。美中不足的是，Module::Starter::Smart 不会自动在测试目录中的 00-load.t 文件中添加该模块的 use_ok 测试条目，所以一定要记得手动添加。

3. 扩展 Module::Starter

如果还是觉得 Module::Starter 无法满足需求，我们可以自己动手写插件。

在内部，Module::Starter 会从左到右依次加载各个插件模块，并动态地让右侧模块继承左侧模块，到达列表尾部时，再追加 Module::Starter 作为最后的继承者，并调用 create_distro 方法最终生成文件目录框架。

一般来说，最后调用的 create_distro 方法来自 Module::Starter::Simple 模块。它会依次调用一系列其他方法完成各项细小任务，所以随后的插件模块可选择重载其中部分方法，达到定制目的。请看 Module::Starter::Plugin::Template 文档，其中列出了部分可供重载的方法。如果有多个插件重载同一个方法，那么位于插件列表最右侧的那个模块始终都能先执行该方法，在完成自己的工作后，它可以呼叫父类（也就是左侧的模块）的同名方法，依次往上回溯整个继承链，或是劫持整个方法。

下面的例子展示了如何在编写生成新模块时自动嵌入企业自己的软件许可证到 README 文件中的插件：

```
package Module::Starter::Plugin::CorporateLicense;
use subs qw/
  _license_blurb_README_license_module_license/;
sub _license_blurb {
return <<'LICENSE';
This program is distributed under the Nameless
Corporate License
This software is developed as is with little regard
to quality. Should this software eat your homework
or take away your birthday, please call our support
line where we will jerk you around until you give
up.
LICENSE
}

sub _README_license {
  my $self = shift;
  my $year          = $self->_thisyear();
  my $license_blurb = $self->_license_blurb();
return <<"HERE";
COPYRIGHT AND LICENCE
Copyright © $year Nameless Corporation
$license_blurb
HERE
}
```

www.TopSage.com

```
sub _module_license {
  my $self = shift;
  my $module = shift;
  my $rtname = shift;

  my $license_blurb = $self->_license_blurb();
  my $year          = $self->_thisyear();
  my $content = qq[
\=head1 COPYRIGHT & LICENSE
Copyright © $year Nameless Corporation
$license_blurb
];

return $content;
}
```

把我们的插件名字写到配置文件~/.module-starter/config 中：

```
plugins: Module::Starter::Simple Module::Starter::Smart ↵
    Module::Starter::Plugin::CorporateLicense
```

以后每次新建模块时，都会自动替换为自己的许可证信息。

4. 使用 Distribution::Cooker

可能你还是会认为像上面那样扩展插件太过麻烦，那么这样，先按照你喜欢的方式新建一个模块，然后修订其内容，做成模板文件。以后要生成新模块框架时，可以让 Distribution:: Cooker 照此依葫芦画瓢。当然，制作模板时可以先用 Module::Starter 搭个粗略的框架。

来自 Distribution::Cooker 的 dist_cooker 命令会从指定的模块模板目录读取各个文件，然后生成实际的模块文件。实际上，dist_cooker 很漂亮地封装了 Template 模块中的 ttree 程序：

```
% dist_cooker
ttree 2.9 (Template Toolkit version 2.20)
Source: /Users/Snuffy/.templates/modules
Destination: /Users/Snuffy/Dev/Perl/Foo
Include Path: [ ]
Ignore: [ \b(CVS|RCS)\b, ^#, (\.git|\.svn)\b ]
    Copy: [ \.png$, \.gif$ ]
  Accept: [ ]
  Suffix: [ ]
...
```

该程序的配置文件~/.dist_cookerrc 用的是 INI 格式：

```
[user]
name = Joe Snuffy
pause_id = bdfoy
email = snuffy@example.com
[license]
perl = 1

[templates]
```

www.TopSage.com

大家网 TopSage.com

```
dir = /Users/Snuffy/.templates/modules
module = lib/Foo.pm
script = bin/script.pl
```

可以看到，模板文件所在目录为/Users/Snuffy/.templates/modules，任何要用的文件都可以扔在里面，dist_cooker 会负责一一处理。此外，dist_cooker 还支持嵌入若干模板变量，比如 module 和 module_file，无需任何设置就可以使用，在模板文件中分别表示模块名称以及所在的文件名称。下面是 README 文件的模板样例：

```
You can install this using the usual Perl fashion
perl Makefile.PL
make
make test
make install

The documentation is in the module file. Once you install
the file, you can read it with perldoc.
perldoc [% module %]
If you want to read it before you install it, you can use
perldoc directly on the module file.
perldoc  lib/[% module_file %]
```

按照模板构建新模块的方式无需编码，又恰如所需，更何况代码越少，出错也越少。

5. 要点

- 用代码生成工具完成重复性劳动。
- 通过插件模块定制 Module::Starter 功能。
- 用 Distribution::Cooker 模块按照既定模板创建新模块。

条款 81　给模块取个好名字

给我们的 Perl 模块取个好名字非常重要。在 CPAN 上，如果模块名字好记且有意义，人们便可以很容易找到它。

名字糟糕的话，就算代码写得无比高明也可能无人问津。可以设想一下，当你发现一个新上传到 CPAN 的模块，一眼看到它的名字，是否可以立即推猜出它的用途？另外，它的名字是否和 CPAN 上已有的其他模块冲突？

关于给包命名这事儿，没有人规定一定要遵循什么准则，甚至连可用作参考的大纲都没有。给模块取什么名字都由你做主。不过，一个好名字总是好处多多。

modules@perl.org（用于 PAUSE[①]系统管理的邮件列表）和 moduleauthors@perl.org 这两个邮件列表对我们如何选择一个好名字会有所帮助。不止如此，它们还提供许多现有模块的命名方式，以便替我们抉择最恰当的命名空间。

① PAUSE 是 The [Perl programming] Authors Upload Server 的缩写，用于提供模块作者发布和维护模块文件到 CPAN 的自助系统，地址为 http://pause.perl.org。——译者注

www.TopSage.com

1. 取名要点

模块名由若干简短字词构成，一旦发布启用后，基本上就不好改动了。名字并不是给自己看的。我们创造了模块，当然很清楚它的用途；但用户缺乏背景了解，乍看下自然不知道它用来解决怎样的问题。所以名字就是用来辅助模块提供相关信息的。基本上，一个好名字应该能点明模块的应用场景、关键功能，并具备区别于其他模块的定义性特征。

- 提供上下文信息

CPAN 名字本身就缺乏足够的上下文信息，理解起来顶多也就是“有关 Perl 的什么东西”。虽然我们在 CPAN 上按类型划分模块，但类型信息本身是游离于模块之外的。用户下载后，或在博客中谈及，或在代码中使用时，都无从了解该模块的类型。假如维护代码时看到这样的模块名字，你会作何感想：

```
use XYZ::WWR::JKL;
```

可能你会觉得这个例子很傻，但事实上，我们确实见过名字中连一个元音都没有的模块，也找不到相关做初始化工作的函数。名字仅对自己或自己所在行业有意义是不够的，更多情形下，它应该让所有人都觉得有意义。毕竟，CPAN 就是这样一个社区，其中各种用途的模块不计其数，名字自然就成了区分它们的重要依据。

模块所能完成的任务和功能特性一般都有应用场景，作者很清楚什么情况下该用此模块，因为多半他们都是为了解决自己的问题而开始创作的。在作者眼里，模块做了些什么以及名字代表的意义，都是显而易见、无需解释的。但其他人不了解这些，而模块的名字理应点明。

比如在 Debian Linux 发行版本里，软件包管理工具称为 dpkg。单从名字来看，没用过 Debian 系统的人根本不知道这是什么东西。但只要谈到 Debian，用过的人就知道说的是软件包管理工具。但如果谈到 Perl，好像没有叫做 dpkg 的东西，所以仍然需要提供更详尽的上下文或语境。

几乎所有缩略词或简写都是意义模糊的。可以试着用 Google 搜索一下缩略词，如果第一页结果展示的内容和该词意义无关，那就说明该缩略词选得有问题。

- 描述关键特性

有些模块用于特定任务，有些则负责处理普通事务。模块的名字应当对此有所说明。一个叫做 HTML 的模块是干什么用的？很难说，对吧。那么 HTML::Parser，HTML::TreeBuilder 以及 HTML::SimpleLinkExtor 呢？基本上我们单靠名称就都能猜出来。所以在选择名字的时候，我们可以参考这些取名方法，尽可能描述清楚模块的关键特性。

- 有所区别

CPAN 上有许多相近模块，或目标相近而实现方式不同，或实现方式相近而目标不同。在 CPAN 上可以找到多少有关 Config 和 Getopt 的模块？能不能光从名字就推断出各自不同之处？如果我们的模块和其他相近模块同在一个命名空间下，该如何区别开来？人们凭什么放弃其他类似模块改用我们的？这都需要有个特别的名字。

2. 部分命名惯例

CPAN 是个大杂烩，也是个自由集市，许多命名方法随着时间推移慢慢发展成为惯例。惯例

www.TopSage.com

也并非一成不变，只要有需求，也必定会持续演进。

- App

我们可以把应用程序按照 Perl 模块的形式发布，而模块名字通常用 App 作为命名空间，比如 App::Ack、App::Cpan 和 App::Prove。这样的名字说明，该模块包含的是可供执行的应用程序，而不是一般用于提供函数或方法的库。

- Local

按照约定，顶级名字空间 Local 是不能和 CPAN 上的任何东西有冲突。这样就保证了我们选择的 Local 之下的名字不会和外界的任何东西发生冲突。[①]

- 活跃项目

有些活跃项目，比如 Moose、DBI、DateTime 和 Catalyst，对命名空间都有一定的约定和划分，以保证各组件能相互配合运行。如果要为这些项目贡献扩展模块，最好到它们的邮件列表上征求一下意见，以求稳妥。

3. 需避免使用的名字

CPAN 从 1995 年开始发展以来，管理员陆续摸索出了一些模块命名的门道，以免使用上的混乱无序。在这个演化过程中，一些好的命名规范逐渐形成，而随着时间的推移，原来那些作为范例的命名方式，到现在来看已经落伍。所以，不要因为以前别人这么用过，就盲目地着用，我们要按照现在大家约定俗成的方式给模块取名。

- 顶级命名空间

一般来讲，我们不建议选用单个顶级命名空间的名字作为模块名称。除非它用于链接其空间下各个模块并提供统一应用界面，或者像 Web 2.0 网站那样凭空捏造一个假想词作为应用程序的名字。可能你会觉得 DB 不错，简单又直观，可以在里面写数据库相关代码。但这样的名字本身并没有说明更多意义。更糟糕的是，碰巧 Perl 的内建调试工具也叫 DB[②]。所以请记住，名字本身应该尽可能提供详细的上下文信息。显然 DB 这个名字很失败（不管是用作数据库还是调试器）。

这也不是说所有单独顶级命名空间的名字都有问题，像一些应用框架，比如 Moose、Catalyst 和 DBI 都是在抽象层面提供一系列方法及函数，用于封装底层的不同实现。它们各成体系，自然也就没有冲突之虞。

- Net

顶级命名空间 Net 是最富争议的一个，最初人们将它应用于各式网络协议的交互实现，像 FTP、HTTP、NNTP 之类[③]。但后来渐渐开始用于协议之外其他网络资源访问，比如与网站之间的交互等。其实它们改至 WWW 或 WebService 命名空间更恰当。所以，当且仅当要实现某种网络协议时才可以用 Net。

- 尽量不要用 Simple、Easy、Reduced、Tiny 等字眼

在 CPAN 上用 Simple、Easy、Reduced 和 Tiny 等字眼应该算是败笔了。这种名字只能暗

① Local 并非 Locale，后者用于本地信息的处理，形似但意义完全不同，不要混淆。——译者注

② DB，其文档中的解释为 DB - programmatic interface to the Perl debugging API，DB 取自 Debug 一词。——译者注

③ FTP、HTTP、NNTP 等末尾的 P 都是协议 Protocol 的意思。——译者注

www.TopSage.com

示出该模块是某个模块的变体，可是，变在哪些地方呢？又变到何种程度呢？为什么需要这样的变化呢？简化后的模块一般都会舍弃或隐藏某些特性，没有原来那么灵活，甚至绝大多数情况仅仅为了迎合作者自己的某项需求而度身定做。那么作者的需求具体又是什么？显然这样的名字无法说明。毕竟，对你来说貌似浅显易懂的名字，对其他人却不尽然。

- *也不要用 API、Interface 之类的名字*

你的模块真是 API 吗？没开玩笑吧？从某种意义上说，模块函数或方法都可以看作是对底层操作的接口，如果不是真的实现某些应用 API 约定的命令和交互方法，就不要在名字里多此一举。

- *不要用自己的名字命名模块*

有许多人想不出合适的概括模块用途的名字，就干脆用自己的名字来命名。就算他的名字很酷，也无法说明模块的确切功用，哪怕后面跟 `Util` 也无济于事。想想看，`Snuffy::Util` 是干什么的？除非这是某个称作 `Snuffy` 的应用程序自带的工具包，否则请不要把自己的名字当作命名空间的一部分。

4. 要点

- 模块名字应当能说明其主要功能。
- 取名应当避免使用模糊的非描述性的词。
- 向 modules@perl.org 咨询合适的命名空间。

条款 82　在代码中嵌入 Pod 文档

许多优秀的软件开发者，特别是在大型项目中，都会约定遵循一套编码规范，比如在每个函数之前写上块注释，描述函数的大体功能、输入输出、预先条件、历史修订等信息。并且，如果这些信息使用统一格式的话，还可以通过脚本提取，自动生成文档。

如此一来，源代码文件就成了自身编码的参考。代码修订时就能立即更新相关文档，不必打开文件另行编辑。这其实也保证了代码和文档的一致性。

在 Perl 源代码中我们可以嵌入 Pod 文档，也就是 Plain Old Documentation 格式的文档。在编译解释脚本时，Perl 会忽略 Pod 部分。但我们可以用 Perl 工具包扫描源码文件，提取 Pod 文档，再转换成 man 手册、HTML 格式网页、纯文本文档等任何格式文件。

1. Pod 基础

Pod 是一种非常简单的标记语言，设计初衷就是要有方便转换到其他格式（文本、HTML 等）的能力。所以，就算再怎样复杂，原始的 Pod 格式文字也不失可读性。

Pod 文档由以空行隔开的段落组成，按照段落所起作用可以分为以下三种：

- *字面文本*

完全按照原样逐字显示的段落，转换到其他格式时不应自动换行，也不做任何标记上的转换处理，通常用于展示代码范例。所以转换后可用不同字体显示，但必须是等宽字体。

www.TopSage.com

● Pod命令

以=字符开头的行，后跟命令标识以及相关可选的文字。目前定义可用的有下面这些，但有些格式转换器可能无法识别其中部分：

- =head1 =head2（一级、二级标题）

```
=head1 Understand Packages and Modules.
=head2 Packages
```

- =item（普通列表或顺序列表中的一项条目）

```
=item 1 ... a numbered item
=item * ... a bulleted item
=item B<NOTE> ... a bolded "other" item
```

- =over N / =back（缩进N个空格 / 结束列表缩进）

```
=over 4
=item * Dog
=back
```

- =pod / =cut（Pod文档的开始 / Pod文档的结束）

```
=pod
=cut
```

- =for X（指定后续段落所属格式为X）

```
=for html
```

- =begin X / =end X（指定格式为X的段落始末）

```
=begin comment
If you can read this, you are using a text translator.
=end comment
```

● 普通文本

除了逐字文本和Pod命令外的，都是普通文本，转换后会自动调整段落宽度，使用便于文字阅读的字体显示。对于行文中加注特殊Pod标记的文字，也会格式化为相应的显示方式：

- *斜体*用 I<text>：

```
You will be I<very> lucky to have John work for you.
```

- **粗体**用 B<text>：

```
You will be B<very very> lucky to have Heather ↵
work for you.
```

- 援引代码片段用 C<text>：

```
now, add 5 to C<$d[$a,$b]>
```

- 强制关闭自动换行用 S<text>：

```
S<foreach $k (keys %hash)>
```

- 嵌入特殊字符用 E<code>：

```
The less-than, E<lt>, is special in Pod
This will be a double quote E<34>
```

- Pod文档的交叉引用 L<text>：

```
L<name>          man page
L<name/ident>    item in man page
```

www.TopSage.com

TopSage.com

```
L<name/"sec">     section in man page
L<"sec">          section in this man page
```

- 表示文件名用 F<lname>：

```
Be careful not to delete F<config.dat>!
```

- 加注索引条目用 X<text>

值得注意的是，有些 Pod 格式转换工具会自动识别函数名（名字后面跟圆括号的）及其他上下文相关格式，然后转成特定方式显示。另外，多数转换工具都会把成对的普通英文双引号转成成对的左右双引号，把两个短划线转成破折号等。

请看下面的 Pod 样例：

```
=head1 My Pod Example
=head2 My 2nd level heading
I<Pod> is a simple, useful markup language for Perl
programmers as well as others looking for a way to write
"Plain Old Documentation."
With Pod, you can:
=over 4
=item 1
Create documentation that can be readily translated into
many different formats.
=item 2
Embed documentation directly into Perl programs.
=item 3
Amaze your friends and terrify your enemies. (Possibly.)
=back
Author: Joseph N. Hall
  Date: 1997
```

用我自己的 pod2mif 转换后的结果：

```
Translated Pod file
My Pod Example
My 2nd level heading
Pod is a simple, useful markup language for Perl
programmers as well as others looking for a way to write
"Plain Old Documentation." With Pod, you can:
Create documentation that can be readily translated into
many different formats.
Embed documentation directly into Perl programs.
Amaze your friends and terrify your enemies. (Possibly.)
Author: Joseph N. Hall
  Date: 1997
```

2. 用 Pod 写 man 手册

轻松转成 man 手册是编写 Pod 格式文档的目的之一，但需额外注意某些约定，以便其他 Unix man 程序正确识别：变量和函数名应使用*斜体*；程序名称，以及命令行选项开关等应使用**粗体**。

此外，man 手册也有既定范式：一级标题应使用全大写字母（CAPITAL LETTERS），并按照以下顺序说明各项常见内容：

- NAME——程序／库／或不管是什么东西的名字。
- SYNOPSIS——简要的模块使用范例。
- DESCRIPTION——详细描述，需要的话可再细分小节。
- EXAMPLES——展示具体的使用样例。
- SEE ALSO——对其他相关模块的手册页索引等。
- BUGS——已知问题或报告问题的方式方法。
- AUTHOR——作者信息、联系方式等。

有关 man 手册详尽的信息组织框架，请参阅 pod2man 文档。

3. 要点

- 用 POD 可以将文档嵌入 Perl 代码。
- 有三种 POD 段落类型：普通文本、字面文本和 Pod 命令。
- 组织文档结构时应符合人们习惯。

条款 83　限制我们的发行版用于特定平台

多数时候，我们都会尽力让自己的模块能在各种操作系统，各个 Perl 版本上运行。但也有例外，像 `Win32::Process` 就只能在 Windows 系统上跑。有时我们用到 Perl 的新特性，在旧版环境中自然无法运行，所以要负责及时报告用户，给出解决方案提示。

无论上传模块到 CPAN 还是内部通过 CPAN 工具链进行分发，一旦出现错误报告，CPAN 工具就会自动作出相应处理。CPAN Testers 网站上有一份写给模块作者的指南（http://wiki.cpantesters.org/wiki/CPANAuthorNotes）可供参考。

1. 使用正确的 Perl 版本

我们熟悉的 `use` 关键字可不仅仅是用来加载模块的。如果给定 Perl 版本号，它就会检查当前运行此脚本的 `perl` 解释器版本是否等于或高于给定的版本号。比如，想用最新的 Perl 5.10 特性的话，需要显式声明：

```
use 5.010;
```

当 `perl` 在编译时看到这句话，就会作版本检查，不符合条件则立即停止编译。之后，它会启用所有 Perl 5.10 的新特性，而这种通过版本号开启或关闭相关特性的能力也是从 Perl 5.10 开始引入的。

另外需要注意，Perl 的版本号由三部分数字组成，小数点后的两个版本数字都使用三位长度。所以人们说的“Perl 5 10 1”，即 Perl 5.10.1，在 `perl` 内部实际上就是 5.010001[①]。

特殊变量 `$]` 保存了当前 Perl 版本号，所以如果想用旧版 Perl，比方说用到的外部模块只能在旧版 Perl 上正常工作，那么可以检查该变量的值，不符合要求就报告错误。一般把版本检查放

① 这就是 `use 5.010` 不能写成 `use 5.10` 的原因。——译者注

www.TopSage.com

在 BEGIN 块中执行，这样程序一开始运行就会进行检查。

看是否为指定的 Perl 版本：

```
BEGIN {
  die "This is perl $], but you need 5.008005"
    unless $] == 5.008005;
}
```

也可以指定版本范围，如 5.10 之前的所有版本：

```
BEGIN {
  die "This is perl $], but you need a version"
  . 'less than 5.010'
    unless $] < 5.010;
}
```

当然也可以指定两个 Perl 版本之间的全部版本：

```
BEGIN {
  die "This is perl $], but you need between "
  . '5.0006002 and 5.008008'
    unless $] >= 5.0006002 and $] <= 5.008008;
}
```

另外，我们也可以在代码中通过这样的方式做选择，或者根据不同的 perl 版本加载不同的模块，灵活应对。

写 Makefile.PL 或 Build.PL 文件时，也可以指定最低能运行的 Perl 版本号：

```
# 在 Build.PL 中的写法
my $mb =
Module::Build->new( ...,
requires => { 'perl' => 5.008001, }, );

# 在 Makefile.PL 中的写法
WriteMakefile( ..., MIN_PERL_VERSION => '5.010001', );
```

2. 检查当前操作系统

除版本检查外，我们也能检查所运行的操作系统类型。最简单的方式是用 Perl 自己提供的特殊变量 $OSNAME（这是全名写法，还有一个等效的简写版本 $^O，见 perlvar 文档）：

```
use English qw($OSNAME);
  die 'Unsupported OS: You have $OSNAME but '
  . "I need Windows!\n"
    unless $OSNAME eq 'MSWin32';
```

这是比较初级的写法，高级一点的可以用 Devel::CheckOS 模块来检查，不光是特定操作系统版本，它还能检查是不是属于某个操作系统家族。像上面例子需要记住具体系统名称，一次也只能检查一种版本，而下面的例子则可以检查是否属于微软的 Windows 家族，且无需理会具体是哪个版本。Devel::CheckOS 文档（http://search.cpan.org/~dcantrell/Devel-CheckOS-1.61/lib/Devel/CheckOS.pm）[①]中引用了 IRC 聊天室[②]中人们的抱怨："$^O 就像那些把裤子当帽子戴还自

① 读者可以使用同一网址的短链接：http://tinyurl/com/4ulffpz。——编者注

② 技术狂热者都热衷于挂在 IRC 聊天室中相互交流，这种习惯从网络发展伊始一直保留至今。——译者注

www.TopSage.com

以为是的家伙，愚蠢且丑陋。”。

```
use Devel::CheckOS qw(os_is);
  die 'OS unsupported! You need Windows to run '
    "this program!\n"
    unless os_is('MicrosoftWindows');
```

这里的OS unsupported消息非常重要：CPAN Testers（见条款97）会在构建模块时根据该信息决定是否报告测试失败。如果只是报告不支持当前系统，可以直接用die_unsupported()函数，它实际上就是die("OS unsupported\n");，虽然无法定制错误消息，但已足够：

```
use Devel::CheckOS qw(os_is die_unsupported);
die_unsupported() unless os_is('MicrosoftWindows');
```

要查看所有操作系统名称，运行下面单行程序：

```
% perl -MDevel::CheckOS -e \
    'print join(", ", Devel::CheckOS::list_platforms())'
```

有些名字看上去怪怪的，甚至还有::出现。这是因为实际上每个可用平台名字背后都是一个Devel::AssertOS::开头的模块，MacOSX::v10_5的名字就来自Devel::AssertOS::MacOSX::v10_5。我们知道模块名字实际是同文件名相关的，所以有些系统名字里的特殊字符也会被拿掉，并按照Perl模块名字要求改写。

3. 检查perl的编译配置

在编译安装perl时（我们自己至少也应该去编译一次，见条款110），编译工具会把所有编译情况和配置内容保存到Config模块。可以运行perl -V查看对应细节。

在我们运行多线程程序之前，首先得确认所用的perl在编译时启用了多线程选项。加载Config模块，便可以访问%Config散列变量，查看当初编译时的配置：

```
use Config;
die "You need a threaded perl to run this program!\n"
  unless $Config{usethreads} eq 'define';
```

如果程序需要64位运算支持，当然也可以对此进行查验：

```
use Config;
die 'You need at least a 64-bit perl to run '
  . "this program!\n"
    unless $Config{ivsize} >= 8;
```

4. 要点

- 如果程序对perl版本有要求，应当检查并限定其所需的perl版本。
- 根据需要限定程序所运行的操作系统以及相关的Perl编译配置。
- 用Devel::CheckOS模块检查当前操作系统类型。

条款84　检查Pod文档

代码可不是唯一需要测试检查的东西，嵌入其间的Pod文档（见条款82）也同样需要仔细检查，以确保格式正确，内容到位。通过测试我们可以发现许多隐含在文档中的错误或遗漏。一般

www.TopSage.com

Perl 程序员会把此项检查放入“作者自己的测试集合”（author tests，见条款 88）。

1. 检查 Pod 格式

编写 Pod 的最终目的是让转换器能正确格式化，生成便于阅读的文档。很少会有人直接翻阅源代码中的 Pod，一般都是查看转换格式后的文本或 HTML 页面。

命令行中对 Pod 文档格式作检查可以用 podchecker 工具：

```
% podchecker program.pl
```

如果遇问题，它会详细报告：

```
*** ERROR: unterminated C<...> at line 6 in file ↵
  program.pod
*** ERROR: =over on line 4 without closing =back (at ↵
  head1) at line 8 in file program.pod
*** ERROR: empty =head1 at line 8 in file program.pod
*** ERROR: =back without previous =over at line 10 in ↵
  file program.pod
program.pod has 5 pod syntax errors.
```

当然也可以在模块测试套件里做此检查。比如在 t/pod.t 测试文件中，使用 Test::Pod 文档中给出的使用范例：

```
use Test::More;
eval 'use Test::Pod 1.00';
plan skip_all => 'Test::Pod 1.00 required' if $@;

all_pod_files_ok();
```

这段代码很酷哦，它会找出所有包含 Pod 文档的文件并依次作语法检查。当然，运行之前得先安装 Test::Pod 模块。

2. 检查 Pod 文档覆盖率

既然Pod是对Perl代码的文档说明，那么保证每个函数文档无一缺漏也是项必不可少的工作。每次给模块新添子程序，都应该一并加上文档说明，不过不用担心是否遗漏，我们可以用 Test::PodCoverage 模块检查覆盖率。一般我们把它放入 t/pod_coverage.t 测试文件：

```
use Test::More;
eval 'use Test::Pod::Coverage 1.00';
plan skip_all => 'Test::Pod::Coverage 1.00 required'
  if $@;
all_pod_coverage_ok();
```

一旦发现没有对应文档说明的子程序，它就会报告错误。它能分辨“私有”子程序并跳过检查。其实我们所说的私有子程序就是名字以下划线开头的那些，传统上我们约定这种形式的名字为私有子程序，但实际上在 Perl 内部是没什么分别的。文档一般是写给模块使用者阅读的，内部实现细节及私有子程序不必向用户说明，所以跳过对这部分的检查是可以的。不仅如此，我们也可以将全大写的子程序名当作私有子程序：

```
use Test::More;
```

www.TopSage.com

大家网 TopSage.com

```
eval 'use Test::Pod::Coverage 1.00';
plan skip_all => 'Test::Pod::Coverage 1.00 required'
  if $@;
all_pod_coverage_ok(
  { also_private => [qr/^[A-Z_]+$/], },
);
```

3. 拼写检查

好像单词拼写从来都不被人重视，但要知道文如其人，如果拼写有错误，怎么说都有些难为情。写得对，那很正常，没人在意，而一旦写错的话，就会被人牢牢记住，留下坏印象。

困难之处在于 Pod 文档穿插在代码之间，普通拼写检查工具无法分辨，八成会把代码当作拼写错误，结果不言而喻。

Pod::Spell 模块就是为解决这个问题设计的，它其实就是个 Pod 格式转换器，会自动跳过代码部分，将文档打印出来，以便交给外部拼写检查工具处理，一般我们用得最多的是 ispell。

假如有段代码和文档，你能一眼看出拼写错误之处吗：

```
use warnings
use strict;

=pod
Before using this module, check the compatability the
public API and the local version the library on your
machine. Hopefully the scripability can help you get it
done.
=cut
sub do_something_amazing {
  ...
}
```

podspell 工具[①]仅输出程序中 Pod 部分的内容：

```
% podspell speling.pl
Before using this module, check the compatability the
public API and the local version the library on your
machine. Hopefully the scripability can help you get it
done.
```

现在便可以将此通过管道传送给命令 ispell 检查拼写错误。上面例子有三处可能出错，ispell 给出了正确拼法的建议：

```
% podspell speling.pl | ispell -a | \
    perl -ne 'print if /^[#&]/'
& compatability 3 36: comparability, compatibility, ↵
  computability
& API 6 61: AI, APE, APT, BPI, DPI, PI
# scripability 57
```

相对来说，拼写检查是比较独立的问题，所以在写 Perl 模块时可以把它放到一个测试文件中，对所有包含 Pod 文档的文件都进行一次拼写检查，利用 Test::Spelling 模块，t/pod_spell.t 可以相当简洁：

① 安装 Pod::Spell 模块后即可使用随附的 podspell 命令。——译者注

www.TopSage.com

```
use Test::More;
use Test::Spelling;
all_pod_files_spelling_ok();
```

4. 要点

- 用 podchecker 命令检查 Pod 格式是否正确。
- 用 Test::Pod::Coverage 模块检查文档覆盖率。
- 用 Test::Spelling 模块作拼写检查。

条款 85　嵌入其他语言代码

有时我们想在 Perl 程序里调用其他语言写的软件库。这在以前并不简单，要么用 Perl 重新实现算法，要么用 Perl 的 XS 机制（见条款 86）将外部库链接进来。但重写实现耗费太大，而且性能也得不到保证，用 XS 的话又很讲究技巧。还算幸运，现在我们可以用 Inline 系列模块直接将其他语言写在 Perl 里面。

来看下面的例子，Inline 模块后端所用的 Inline::Java 结合了 Perl 的快速脚本编程能力和 Java 的统计库功能，由 Perl 负责网络资源获取、数据预处理等工作，然后交给 Java 的统计分析工具包 Classifier4J 返回最终结果：

```
use LWP::Simple;
use Inline (
Java => <<'END_OF_JAVA_CODE'
import net.sf.classifier4J.summariser.ISummariser;
import net.sf.classifier4J.summariser.SimpleSummariser;
import net.sf.classifier4J.ClassifierException;
public class MySummarizer {
  private ISummariser summarizer;
  public MySummarizer() {
    summarizer = new SimpleSummariser();
  }

  public String summarize(String input, int sentences)
    throws ClassifierException {
    return summarizer.summarise(input, sentences);
  }
}
END_OF_JAVA_CODE
, CLASSPATH => 'Classifier4J-0.6.jar');
my $input = get(
    'http://www.constitution.org/cons/constitu.txt');
$input =~ s/\[[^]]*\]/ /g;  # 去除方括号及其中的文字
$input =~ s/\s+/ /g;        # 将连续多个空格白字符合并为一个空格
print MySummarizer->new->summarize($input, 1), "\n";
```

上面这段代码先是下载一份美国联邦宪法，清理部分文本后，交由 Java 类 MySummarizer 里的 summarize 方法进一步处理：

```
% perl summary.pl
```

www.TopSage.com

```
All legislative Powers herein granted shall be vested
in a Congress of the United States, which shall consist
of a Senate and House of Representatives.
```

嵌入其他语言代码的用处不止是引用外部软件库这么简单，在需要高性能计算的时候它同样有用。比如检查某个数字是否为质数，用 Perl 来做的话极为耗时：

```
sub is_prime {
  my $number = shift;
  my $divisor = int( sqrt $number ) + 1;
  while ( $divisor > 1 ) {
    return 0 if $number % $divisor-- == 0;
  }
  return 1;
}

my $prime_count = 0;
my $ceiling     = 1000000;

foreach ( 1 .. $ceiling ) {
  $prime_count++ if is_prime($_);
}
print "There are $prime_count primes under $ceiling\n";
```

如果改用 Inline::C，我们会发现显著的性能提升：

```
use Inline qw(c);
my $prime_count = 0;
my $ceiling     = 1000000;

for ( 1 .. $ceiling ) {
  $prime_count++ if check_prime( $_, int( sqrt $_ ) + 1 );
}

print "There are $prime_count primes under $ceiling\n";
__END__
__C__
int check_prime(int number, int divisor) {
  while (divisor > 1) {
    if (number % divisor-- == 0) {
      return 0;
    }
  }
  return 1;
}
```

看看，光这么一个简单的例子，就有了如此显著的性能改善！

```
% time perl prime.pl
There are 78498 primes under 1000000

real    5m40.985s
user    5m28.979s
sys     0m1.462s
```

www.TopSage.com

```
% time perl prime_inline.pl
There are 78498 primes under 1000000

real    0m14.295s
user    0m12.223s
sys     0m0.794s
```

当 Perl 运算成为程序性能瓶颈，用 C 改写关键代码后，其性能一般都会有显著提升。此外，还可以到网上找现成的算法更优的代码来直接用，进一步改善性能。

要点

- 用 Inline 模块嵌入其他语言代码，以便使用第三方软件库。
- 在为了优化性能而嵌入其他语言之前，先改进我们的 Perl 程序算法。

条款 86　用 XS 链接低级语言，提高运行速度

XS 是一种胶水语言，能将 Perl 同 C 或 C++ 代码链接起来。它通过一个特殊的预处理器来帮我们搞定数据类型转换及其他细节工作。

有了 XS 的链接能力，我们可以用 C 或 C++ 来写函数，然后在 Perl 脚本里调用。实际上 Perl 调用的是 XS 里封装过的子程序，我们称为 XSUB。在 XSUB 里我们可以完整地对 Perl 的内部进行访问，并可以执行任何我们需要的功能，像创建变量，修改变量值，运行 Perl 代码等。

过去我们用 XS 实现混合语言编程的地方，现在多半可以用 Inline 系列模块（见条款 85）以更优雅的方式轻松实现。所以在实际应用中，尽可能优先尝试用 Inline 的方式集成混合语言代码。

用 XS 写成的模块实际上是动态加载的共享库文件。从头开始写 XS 模块的话有数不尽的琐碎细节需要操心，不过对于 XS 接口我们引入了一大堆相关的辅助工具，帮我们处理这些麻烦。多数应用场合都比较简单，完全可以用 x2hs 工具自动生成模板文件，然后只需往里添加相应的 Perl 及 XS 代码，最后运行 make 编译为共享库，测试通过后构建软件发行版进行发布，供最终用户安装使用。

XSUB 提供了在 Perl 里访问操作系统所支持特性的能力，比起 syscall 来更胜一筹。我们也可以将那些耗时很多、需要反复多次运行的子程序迁入 XSUB，以提高运算速度。当然，借助 XSUB 我们也可以给现有的 C 或 C++ 代码库添加 Perl 模块接口，供 Perl 脚本调用。

假如我们需要一个子程序，随机打乱给定列表中各元素后生成新的列表返回。用 Perl 实现的话，可能会用到 Fisher-Yates 混排算法，其实现代码大体如下：

```
sub shuffle2 {
  my @result = @_;
  my $n      = @result;
  while ( $n > 1 ) {
    my $i = rand $n;
    $n--;
    @result[ $i, $n ] = @result[ $n, $i ];
  }
  @result;
}
```

www.TopSage.com

如果运算的效率对解决问题非常重要，Perl通常不是最佳选择。因此，我们可以在XS里将它重写，让它按C语言的速度运行。

1. 生成模板文件

开始编写自己的混排C实现之前，先用h2xs自动生成模板文件：

```
% h2xs -A -n List::Shuffle
Writing List-Shuffle/ppport.h
Writing List-Shuffle/lib/List/Shuffle.pm
Writing List-Shuffle/Shuffle.xs
Writing List-Shuffle/Makefile.PL
Writing List-Shuffle/README
Writing List-Shuffle/t/List-Shuffle.t
Writing List-Shuffle/Changes
Writing List-Shuffle/MANIFEST
```

如上所示，工具h2xs创建了一系列胶合C语言时必不可少的文件。

2. 编写并测试XSUB函数

一般XS源代码文件使用.xs作为后缀，XS编译工具xsubpp负责把XS转换为C代码，并生成Perl调用接口。但我们不必亲手运行xsubpp，相关的编译任务会自动集成到Makefile里，我们只需要执行make即可。

XS源代码的开头是一段作为序言的C代码，接着是一个MODULE指令，指定Perl的名字空间：

```
MODULE = List::Shuffle PACKAGE = List::Shuffle
```

上面这行代表真正XS源代码的开始。在实现具体混排操作之前，先用一个简单的XUSB看下效果，该XSUB调用C语言标准库的log函数返回计算结果：

```
double
log(x)
  double x
```

首先第一行是返回数据的类型（double），数据类型需单独成行，接下来一行是函数名称（log）和参数名称列表（(x)）。紧跟着返回类型和函数名称之后的语句一般来说都会缩进，以方便阅读。在本例中，XSUB的正文部分只有一句，即单行声明x为double类型数据。

由xsubpp生成的代码会创建能在Perl中调用的C的同名子程序log()。所生成的代码还包括将Perl参数转换为C double类型的部分，以及返回结果的数据类型转换的部分。百闻不如一见，请看下xsubpp生成的C代码，不用担心，自动生成的代码完全具备可读性[①]。

下面的例子稍微复杂些，调用的是UNIX上的realpath()函数（并不是所有系统上都有此函数）：

```
char *
```

① 编辑Shuffle.xs后保存，运行perl Makefile.PL生成编译用的Makefile文件，然后运行make生成Shuffle.c，打开此文件查看。——译者注

www.TopSage.com

TopSage.com

```
realpath(filename)
  char *filename
  PREINIT:
    char realname[1024]; /* 或用 MAXPATHLEN 代替具体数字 */
  CODE:
    RETVAL = realpath(filename, realname);
  OUTPUT:
  RETVAL
```

生成的 Perl 函数接受一个字符串参数（文件名），返回数据也是字符串类型数据（文件路径）。XS 负责处理 Perl 字符串到 C 的 char * 类型的相互转换，所以在 Perl 代码中使用该函数和使用原生的 Perl 子程序没什么两样：

```
my $realname = realpath($filename);
```

CODE 部分调用了实际进行运算的 C 函数，PREINIT 部分则为 CODE 中用到的变量预先作好声明。变量声明应该放在 PREINIT 部分，而不是和 CODE 中的代码混在一起。RETVAL 是 xsubpp 提供的特殊“魔力”变量，它保存 C 函数返回的计算结果。最后的 OUTPUT 部分则列出 XSUB 函数要返回的数据，一般至少应包含 RETVAL，当然另外也可以一并返回其他数据，如被修改过并从函数中返回的输入参数，其效果就像传递引用的函数调用一样。

上面的例子都返回单一标量值，但我们要做的混排函数是要返回标量元素列表，所以必须得用 PPCODE 自己完成对参数和返回值的出栈（pop）和压栈（push）操作。听起来好像有点难，实际并不复杂。好吧，让我们开始动手，把下面代码写入 Shuffle.xs 文件 MODULE 那行后面：

```
PROTOTYPES: DISABLE
void
shuffle(...)
  PPCODE:
  {
    int i, n;
    SV **array; /* SV 是标量值 (scalar value) 类型 */
    SV *tmp;
    /* 分配内存 */
    array = New(0, array, items, SV *);
    for (i = 0; i < items; i++) {
    /* 复制参数 */
    array[i] = sv_mortalcopy(ST(i));
  }
  n = items;
  while (n > 1) { /* 按 Fisher-Yates 混排算法打乱 */
    i = rand() % n;
    tmp = array[i];
    array[i] = array[--n];
    array[n] = tmp;
  }
  for (i = 0; i < items; i++) {
    XPUSHs(array[i]); /* 将结果依次压入栈 */
  }
  Safefree(array); /* 清空内存 */
}
```

开头的 PROTOTYPES: DISABLE 指令表示在之后的 XSUB 代码中关闭 Perl 函数原型处理[①]。

这里的大概思路是把输入参数依次存入临时数组，然后混排，最后将结果压入栈返回。参数类型为标量元素列表，在 Perl 内部使用类型 SV * 表示。

和 CODE 块不同，使用 PPCODE 的话 XSUB 会关闭对栈中值的自动处理。所以我们可以用 PPCODE 自行定义要返回的内容及形式。参数数量保存在特殊变量 items 里，而参数保存在 ST(0)、ST(1) 等宏定义的指针变量中。[②]

栈中的 SV 指针指向传递给 shuffle() 函数的参数值，我们要的是它的副本，所以后面用 sv_mortalcopy() 函数（会增加引用计数）把传入的标量数据依次复制到数组中。

该数组由 Perl 内部的 New() 函数分配内存，标量数据压入该数组后，按混排算法打乱顺序，依次通过 XPUSHs() 函数把结果放到 XSUB 的栈上去，最后释放存放指针的临时数组。以上我们谈得非常粗略，如果有什么不明白，还请翻阅 perlguts 和 perlxs 手册。

现在让我们保存 Shuffle.xs 并构建此模块：

```
% make
```

编译 Shuffle.xs 和构建其他任务的过程会显示几行看似杂乱的输出，不用理会它们。完成之后，我们来创建一个测试脚本，打开测试文件模板 t/List-Shuffle.t，添加下面代码：

```
# List-Shuffle.t
use Test::More tests => 2;
BEGIN { use_ok('List::Shuffle') }

my @shuffle = shuffle 1 .. 10;
diag "@shuffle";
pass 'shuffled';
```

保存后运行测试。可以看到测试通过，并随之输出诊断信息及混排结果：

```
% make test
...
t/List-Shuffle.t .. 1/2 # 4 9 6 1 3 5 7 2 10 8
t/List-Shuffle.t .. ok
All tests successful.
```

3. 要点

- 用 XS 创建调用 C 代码库的 Perl 接口。
- 用 Inline 模块创建调用其他语言代码库的 Perl 接口。
- 使用现成的其他语言代码库，不必在 Perl 中重复发明轮子。

① 形如 sub myfunction ($$;$) 的函数，其中的 $$;$ 即为参数原型定义。请参阅 **perldoc perlsub** 的 **Prototypes** 部分。——译者注

② 每个 XSUB 函数都有一个栈（Stack），输入参数和返回数据都保存在该栈中。而 ST(x) 是栈中 x 位置上内容的指针。——译者注

www.TopSage.com

第10章 测试 10

Perl 的测试文化着实令人惊叹，它有一整套工具专门用于代码的测试和管理。感谢 CPAN Testers（见条款 97），可能这个星球上测试得最多最广泛的就是 Perl 代码了。

第一个模块发布的时候 Perl 就有了自己的测试架构，而真正崭露头角是从 Michael Schwern 发布 Test::More 模块开始。这个模块把测试工作变得如同折纸一般轻松，因为只需加载测试模块，写上几句简短直白的函数即可：

```
use Test::More tests => 2;

ok( $some_value, "The value is true" );
is( $got, $expected, "The values are the same" );
```

Test::More 模块提出了 TAP（Test Anywhere Protocol，通用测试协议）的概念。该协议约定了一种简单的测试结果输出格式规范。此后不光是 Perl，其他语言也逐渐开始采用这种规范。

得到的测试结果由 Test::Harness 模块负责分析处理，并提供概述性报告。这项工作隐藏在 Test::More 幕后，平时并不为人注意。

之后 Perl 社区中的 Birmingham Perl Mongers 小组发起建设 CPAN Testers 项目，从志愿者手里收集大量 CPAN 模块的测试报告。现如今，该项目已经收集了几百万份可供公开浏览的测试数据。

与此同时，Perl 测试相关的实践和技术也随之演进，日趋完善。Perl 开发者不再局限于测试 Perl 代码，其他诸如发布版本的各项细节，包括文档格式和拼写，甚至测试脚本本身，都可以进行测试以保证质量。

不要觉得编写测试属于额外多余的工作，因为一旦准备好测试组件，其后用在查找程序错误上的时间就会少很多，而省下来的时间就可以用来增加新特性和完成待办任务列表上的项目。所以，还等什么呢，让我们现在就开始测试吧。

条款 87 用 prove 灵活运行测试

我们为代码所写的测试脚本本身也都是 Perl 程序，既然测试脚本是 Perl 程序，自然可以使用 perl 直接运行：

```
% perl t/my_test.t
```

www.TopSage.com

需要留心的是测试脚本中所加载的被测试模块的版本。说不定，我们正开发和测试的模块，其实已经被安装到系统中了。毕竟我们写模块的目的只是为了使用它，对吧？

一如既往，perl 搜索 @INC 中的路径寻找需要加载的模块，先在哪个位置找到就先加载谁。所以，究竟加载的是之前业已安装的稳定版模块还是当前我们要进行测试的开发版模块，取决于 @INC 中的路径设置。

那么，如何确保测试脚本中总是加载开发版本呢？

一种简单的办法，在命令行显式声明将包含新版代码的 blib/ 目录作为优先搜索路径：

```
% perl -Ilib t/my_test.t
```

另一种办法，如果模块使用了自动构建脚本，则可以用名为 blib 的模块。blib 模块会在当前路径下，及若干级[①]上级目录中，搜索名为 blib 的目录（该目录一般由构建脚本自动创建，并把模块代码置入其中），把找到的第一个 blib 目录的路径压入 @INC 起始部分，由此优先加载使用该路径下的代码：

```
% make; perl -Mblib t/my_test.t

% ./Build; perl -Mblib t/my_test.t
```

如此就不用担心弄错测试版本了，接下来运行测试。总共有 500 项测试的话，每条测试结果都会打印一行，在屏幕上会飞快地滚动：

```
1..500
ok 1 - testing some code
ok 2 - another test
ok 3 - let's see if this works
ok 4 - everything should be okay here
ok 5 - checking some functionality
...
ok 500 - this should be looking good
```

这样很容易错过其中失败的项目。因此，perl 提供了一个叫做 prove 的趁手好工具，它负责解析与上面格式类似的测试输出，然后仅打印总结性的测试结果报告。其 -b 开关，就相当于之前的 -Mblib，将 blib/ 目录压入 @INC 头部优先搜索被测试模块：

```
% prove -b t/my_test.t
my_test.t .. ok
All tests successful.
Files=1, Tests=500, 0 wallclock secs ( 0.09 usr 0.01 ↵
   sys + 0.09 cusr 0.00 csys = 0.19 CPU)
Result: PASS
```

如果某项测试没有通过，prove 会给出详尽信息：

```
% prove -b t/my_failing_tests.t
my_failing_tests.t .. 1/500
#   Failed test 'checking some functionality'
#   at my_failing_tests.t line 9.
```

① 最新的 Perl 5.12.1 内建的 blib 模块会查找五级。——译者注

www.TopSage.com

```
# Looks like you failed 1 test of 500.
my_failing_tests.t .. Dubious, test returned 1 (wstat 25↵
   6, 0x100)
Failed 1/500 subtests

Test Summary Report
-------------------
my_failing_tests.t (Wstat: 256 Tests: 500 Failed: 1)
Failed test: 5
Non-zero exit status: 1
Files=1, Tests=500, 0 wallclock secs ( 0.08 usr 0.01 sy↵
   s + 0.08 cusr 0.00 csys = 0.17 CPU)
Result: FAIL
```

仅从解析并总结测试结果这一功能来看，prove 已算相当不错的工具了；而实际上，它能做的还有很多。

1. 随机化测试顺序

prove 的一项极为有用的特性就是打乱测试脚本的运行顺序。有时，测试脚本之间潜在的数据相依性会让测试看起来都能正常通过，而打乱顺序运行或许能发现其中的问题。用 --shuffle 选项启用该特性：

```
% prove --shuffle
```

而 prove 的另一项很棒的特性则是允许并行测试，可以用它的 -j 参数指定同时运行的测试脚本数量[①]。显然，并行测试能节约很多时间。

prove 还具备保存运行状态信息的能力。通过启用 --state 选项，保存的状态信息可用于下次运行时诊断测试失败的项目。

比如在随机顺序测试中，因为每次顺序不同，虽然发现测试失败，但重复运行时未必能再现那个错误。而如果保存状态信息的话，就能按原来顺序执行，重现错误：

```
% prove --shuffle --state=save
……某些失败测试的输出……
% prove --state=last t/*t
```

除此之外，它还可以指定仅运行上次测试时失败的或成功的测试，也可以按不同顺序运行：从运行得快的到慢的，或者反过来从慢到快，或者先测试新增的脚本，或者仅仅运行上次测试以来修改过的脚本。这样一来，诊断错误时就不必每次都从头到尾完完整整地跑一次，而只需关注有问题的部分就行了。

2. 其他语言也能用 prove

prove 只是解析测试结果的输出，所以究竟是 Perl 写的测试脚本，还是测试的是 Perl 代码，它都无所谓。只要符合 TAP 规范的测试结果输出，都可以用 prove 解析。先前例子中我们看到的测试输出结果，就符合 TAP 规范。

① 或者说，同时运行的任务（job）数量。——译者注

www.TopSage.com

大家网 TopSage.com

只需用 --exec 指定运行测试脚本的命令（通常为语言对应的解释器）即可。当然，测试脚本得按照 TAP 规范输出结果，才能为 prove 所用。比如运行 ruby 写的测试脚本可以这样：

```
% prove --exec '/usr/bin/ruby -w' t/*t
```

3. 要点

- 关注于某项特性时，应单独运行该项测试。
- 可以用 prove 有选择地运行一组测试。
- 可以用 prove 测试其他语言编写的测试脚本。

条款 88　有目的性地运行测试

Perl 模块在创建之初，所有测试文件默认放置于 t/目录下，每次编译时都会运行所有测试脚本。而换用 prove 的话就不必这样（见条款 87）。

那么多测试中，有些仅用于确认要实现的功能是否完备，有些针对于特定情形特殊情况,各有侧重。所以有时候，我们只需运行一小部分测试，确认当前环境下能正常运行就可以了。

1. 针对作者的测试

并非所有测试都面向程序功能，有些仅是用来为模块作者自己服务，比如对代码内文档的测试（见条款 84），对测试脚本本身的测试（见条款 96），以及技量（kwalitee）测试（见条款 68），等等。和保证模块功能正常的测试不同，这类测试没那么重要，即便测试失败也可以允许用户安装。

● 使用 TEST_AUTHOR 环境变量

要在别人的机器上运行测试时跳过针对作者的测试，最常见的办法是在自己的机器上设置类似$ENV{TEST_AUTHOR}这样的环境变量[①]，然后在测试脚本中作相应处理：

```
use Test::More;
plan skip_all =>
  'Set $ENV{TEST_AUTHOR} to enable this test.'
  unless $ENV{TEST_AUTHOR};
eval "use Test::Pod 1.14";
plan skip_all => 'Test::Pod 1.14 required' if $@;

all_pod_files_ok();
```

请注意，TEST_AUTHOR 这个名字很常见，其他模块作者也经常用。他们编译测试模块时，看到这部分测试或许会有些惊讶。同样的，我们或许也会遇到这种情况。

我想你一定猜到了解决办法，把 $ENV{TEST_AUTHOR} 设为自己名字，然后在程序中检查：

```
plan skip_all =>
  'Set $ENV{TEST_AUTHOR} to enable this test.'
  unless $ENV{TEST_AUTHOR} eq 'SNUFFY';
```

不过仍然有些小问题，团队合作一起写这个模块的话，别人怎么运行作者相关的测试呢？只

① 显然，环境变量的名字随你高兴取什么都可以。这里用 TEST_AUTHOR 只是便于说明问题。——译者注

www.TopSage.com

好假扮你了！有点遗憾。[1]

- 使用 xt/目录

与其混在一起，不如把作者自己的测试单独抽取出来，放到别的目录中去。一般大家会用 xt/这个名字，表示**额外的测试**（extra test）。需要的时候用 prove 运行该目录下的测试脚本即可（见条款 87）。

如果不想把这些测试随同模块一起发布出去（当然首先我们得分离到其他目录），可以在 MANIFEST.SKIP 文件中标注一笔，它和 MANIFEST 文件一样允许使用 Perl 正则过滤，跳过 xt/ 目录：

```
^xt/
```

如果还是混在一起，那就只好逐个列出这些文件，写到 MANIFEST.SKIP 里去了，记得每次新增测试脚本都要同步更新：

```
t/pod.t
t/pod_coverage.t
```

2. 根据需要运行特定测试

跳过操作系统相关测试和跳过作者测试基本类同，无非是把环境变量改为 Perl 的特殊变量 $^O 进行检测罢了：

```
use Test::More;
plan skip_all => 'Skipping Linux-only tests'
  unless $^O eq 'linux';
```

用 skip_all 会跳过整个测试脚本，有时只需跳过其中部分项目，可以把这部分放到以 SKIP 为标签的块中：

```
SKIP: {
  skip 'Skipping Linux-only tests', 2
    unless $^O eq 'linux';

  ok(...);
  is(...);
}

ok( 1 == 1, 'The universe is stable' );

SKIP: {
  skip 'Skipping thread-only tests', 2
    unless $Config{'usethreads'} eq 'define';

  ok(...);
  is(...);
}
```

此外，限制模块用于特定平台（见条款83）中的办法也同样适用于限制特定平台上运行的测试。

[1] 其实，根本不是什么大问题，加个组名进去用正则匹配检查就行。比如 chunzi@effective-perl-2e。——译者注

www.TopSage.com

3. 自动化编译时跳过测试

CPAN Testers（见条款97）会自动测试任何上传到CPAN的模块。整个过程都是程序化实现的，运行测试的时候可没人盯着屏幕看结果。所以有些测试，比如图形界面弹出窗口期待用户点击按钮确认，在全自动环境下显然无法继续，应该跳过。可以在 SKIP 标签块内检查环境变量$ENV{AUTOMATED_TESTING}是否设置：

```
SKIP: {
  skip 'These tests needs a real person', 2
    if $ENV{AUTOMATED_TESTING};

  test_start_window();
  test_button_push();
}
```

4. 要点

- 把针对作者自己的测试脚本分离到 xt/目录。
- 用 SKIP 标签块跳过一组测试。
- 检查环境变量$ENV{AUTOMATED_TESTING}以跳过非交互式测试环境。

条款89　用依赖注入避免特殊测试逻辑

在生产服务器上，运行中的程序理应连接生产数据库，但测试脚本不能。而在持续构建过程中，也许我们不希望真的有邮件被发出去（见条款98）。总之不管什么情形，最简单最自然的办法似乎应该是给代码加上额外的条件逻辑内容，但，很遗憾，这么做不对：

```
my $dbh =
  ($test_mode)
  ? connect_to_mock()
  : connect_to_prod();

if ( not $testing ) {
  send_nag_email();
}
```

我们每个人都这么干过，就算你不想承认。看上去好像没什么问题，但实际上却制造了麻烦。条件分支语句造成了一次只能测试其中一根分支的状况，这就限制了代码测试覆盖率（见条款96）。[①]如果忘了检查代码而进行测试，一旦失误发送大量无用邮件给最终客户，只会让人难堪。

当然，解决办法还是有的，既可以避免使用条件分支，又能适应不同情境。**控制反转**（inversion of control）就是一种比较优雅的办法，具体一点，我们称之为**依赖注入**[②]（dependency injection）。依赖注入的外延比较宽泛，不光是在测试领域，一般编程实践中也有广泛使用。一旦对此有所体

① 想想看，嵌套分支一多，怎么能保证所有测试代码都至少运行一次呢？——译者注

② 依赖注入（dependency injection），实际上是一种编程模式，把所依赖的资源交给别人，由其负责提供，自己成为资源消费者，也就是（把）依赖（的东西）注入（进来使用）。——译者注

www.TopSage.com

会领悟，以后在你写程序时也会开始喜欢用依赖注入这种编程方式。

该方法形式上很简单，将对象所依赖的部分提取到对象外部即可，而不是在对象内部自产自销。相应的功能由外部模块或对象负责实现，而对象本身仅仅是取用它们。

下面是类 Foo 的对象构造方法，创建对象时接受数据库连接参数，然后创建数据库控制句柄，并缓存到对象内部，最后返回 Foo 对象变量：

```
package Foo;

sub new {
  my ( $class, $dsn, $user, $password, $parameters ) = @_;
  my $self = bless {}, $class;
  $self->{dbh} =
    DBI->connect( $dsn, $user, $password, $parameters );
  return $self;
}
```

采用依赖注入的办法，取消自己创建 DBI 连接，所需做的仅仅是改为接受已经准备好的数据库控制句柄：

```
package Foo;

sub new {
  my ( $class, $dbh ) = @_;
  my $self = bless {}, $class;
  $self->{$dbh} = $dbh;
  return $self;
}
```

貌似平淡无奇，但这么做确实增加了代码灵活性。对于第一种写法作测试的话，必须提供真实可连接的数据库参数，实际连上去测试；但第二种写法则允许使用任何类型的数据库连接，可以和前面一样是真实的测试数据库，也可以是冒名的数据库连接对象（见条款 92）。不管它究竟是什么，只要让 Foo 觉得这是可用的数据库连接就成。

人们常喜欢将依赖注入结合工厂设计模式一起用。下面是由工厂模块负责构建数据库连接对象的例子：

```
package DatabaseFactory;

sub new {
  my ( $class, $dbh ) = @_;
  my $self = bless {}, $class;
  $self->{dbh} = $dbh;
  return $self;
}

sub get_dbh {
  my ($self) = @_;
  return $self;
}

sub set_dbh {
  my ( $self, $dbh ) = @_;
  $self->{dbh} = $dbh;
```

www.TopSage.com

```
}

package Foo;

sub new {
  my ( $class, $database_factory ) = @_;
  my $self = bless {}, $class;
  $self->{database_factory} = $database_factory;
  return $self;
}

sub do_something {
  my ($self) = @_;
  my $dbh = $self->{database_factory}->get_dbh();
}
```

而在测试时，我们可以创建一个冒名数据库链接给它，完全不用修改被测试代码，也不用改变条件语句分支，一切都很自然：

```
my $dbh     = create_new_mock_database_handle();
my $factory = DatabaseFactory->new($dbh);
my $foo     = Foo->new($factory);
$foo->do_something();
```

这样不但用上了依赖注入，还给代码增加额外的逻辑层。这样就使得我们不但可以从对象工厂中返回对象，还能在运行中的任意时刻对对象本身作出修改：

```
my $dbh     = create_new_mock_database_handle();
my $factory = DatabaseFactory->new($dbh);
my $foo     = Foo->new($factory);
$foo->do_something();
$factory->set_dbh(
  create_modified_mock_database_handle() );
$foo->do_something();
```

随着应用扩展，工厂模块可能很快就会变得笨重不堪。特别是当我们的应用变得庞大起来，又期望保留依赖注入带来的好处时，不妨考虑使用另一个控制反转的框架 `Bread::Board`。

要点

- 不要在代码中编写判断是测试环境还是生产环境的逻辑。
- 把依赖的对象提取到外部，通过参数传入使用。
- 用工厂模式创建依赖对象。

条款90 避免给方法引入不必要的东西

单是用了依赖注入（见条款89）还不够，保持依赖最小化也很重要。一般来讲，超过两个依赖就会让代码变得难以维护。如果每个方法都这样，岂不是让人抓狂？

来看一个检查驾照是否过期的模块，假设这段 Perl 代码安置于交通信号灯边的自动相机里。该模块本身并不代表汽车，但它可以借用表示汽车的对象来访问驾照信息：

www.TopSage.com

TopSage.com

```
package Auto::LicenseChecker;
# 也可叫做 City::RevenueGenerator ①

use DateTime ();

sub new {
  my ( $class, $automobile ) = @_;
  my $self = bless {
    automobile => $automobile
  }, $class;
}

sub check_expired_license {
  my ($self) = @_;
  my $licenseplate =
    $self->{automobile}->get_licenseplate;

  return $licenseplate->get_expiration <
    DateTime->new->now();
}
```

上面代码中，构建 Auto::LicenseChecker 对象需要传入整个汽车对象。而在测试脚本中构建 Automobile 时，有可能需要预先设置汽车的各种不同部件或属性信息，可我们现在需要的仅仅是驾照信息：

```
use Tes::More tests => 1;

my $automobile = Automobile->new(
  engine          => Engine->new,
  left_front_tire => Tire->new,
  ...
  license_plate   => LicensePlate->new($number),
  ...
);

my $checker = Auto::LicenseChecker->new($automobile);
ok( not $checker->check_expired_license,
  'legal to drive');
```

显然，要做的事不少。只是为了检查驾照有效期就如此大动干戈，未免有些夸张。自动相机不会在乎车子有几个轮子，或者用的是什么引擎，它只想搞明白经过路口的是谁的车，他的驾照有没有过期，仅此而已。所以只要给它表示驾照的对象就够了：


```
package Auto::LicenseChecker;

use DateTime ();

sub new {
  my ( $class, $licenseplate ) = @_;
  my $self = bless {
    licenseplate => $licenseplate,
```

① 这里是作者的小幽默，他称之为城市税务创收的来源。——译者注

www.TopSage.com

```
  }, $class;

}

sub check_expired_license {
  my ($self) = @_;
  return $self->{licenseplate}->get_expiration <
    DateTime->new()->now();
}
```

忽略掉那些无关紧要的细节，这样在测试中所需的设置工作就大大精简了：

```
use Test::More tests => 1;

my $plate = LicensePlate->new($license_plate);

my $checker = Auto::LicenseChecker->new($plate);
ok( not $checker->check_expired_license,
  'legal to drive' );
```

实际上，如果测试代码复杂，那就意味着应用该模块的代码也必定同样复杂。所以，一旦看到测试脚本中有繁复的初始设置，就该检查一下代码适当做些优化。

要点

- 避免给方法引入不必要的东西。
- 测试代码繁杂，意味着使用该模块的代码也必定繁杂。

条款91　把程序写成模块便于测试

许多Perl程序员都喜欢用短短几行代码快速解决当前问题。既然是临时派用场的，写得潦草些也没什么。可当有一天某个家伙跑来要你改进它完善它，那么最终它会慢慢演变成庞大而复杂的程序，甚至开始影响到整个公司的日常运营，要是这个时候程序出什么差错，你显然难辞其咎。

这种程序演变到最后，往往很难划分成块，分别测试。常见的Perl程序运行起来都是通首至尾，要测试某一点，就得从头到尾运行一遍。此时我们所做的任何事，如打印输出或警告信息，都难以分开捕获测试，也就难以保证程序质量。

解决办法是把程序当成模块来写，而事实上这些模块很可能最终只是像普通程序一样运行。这类程序我们称之为“modulino”，因为编写之初的程序很小，所以也叫做“小模块”。

这个思路的关键是，让它在运行时像程序，而在测试时像模块。也就是说，作为程序运行时，和原来一样一步步完成任务；作为模块在测试脚本中加载时，则仅仅编译而不运行任何代码。

最简单的实现方式，是检查caller的返回值。如果当作模块加载，必定有个更高级别的调用者，也就是加载该模块的测试脚本，那么caller返回真。如果当作程序运行，就没有什么所谓的调用者，caller返回假。以此来决定其行为：

```
package MyApplication::Icelandic;

use warnings;
```

www.TopSage.com

```
use strict;
use utf8;

MyApplication->run unless caller;

sub run {
  print "Starf þitt byrjar hér.\n";
}

1;
```

乍看起来和一般的 Perl 程序很像，但我们把所有要运行的代码都放到了 run 函数里面，它有点像其他语言中的 main 函数。不过这里叫什么名字无所谓，run 也好，main 也罢，activate 也行，只要你喜欢。只需将其中可立即执行的部分隔离在子程序中，那么我们便可以缓一缓，在万事俱备的情况下再调用它以让程序真正工作起来。这种方式便是 modulino。在语言的演进中，Perl 本身为了节省代码书写量而把整个文件都当成 main 函数，而这种 modulino 方式倒是有点开倒车了。

现在可以完善代码了。注意力要放在如何分离各部分功能，写成便于测试的函数，而不是一股脑儿往 run 里面加东西。

回到上面的例子，现在如果要测试该 modulino 的话，还有些困难。其中，run 方法打印的消息是一句固定的话，而且始终输出至默认的文件句柄。尽管这么点代码的确只能干那么点事儿，但它和普通程序存在的问题一样，难以进行测试。

通过将输出的目的地从 run 里面独立出来，我们可以帮助测试脚本顺利捕获输出信息。要实现这一功能，可以创建一个应用程序对象，然后把输出消息时所打印的句柄通过对象属性来指定：

```
package MyModulino;

use strict;
use warnings;

__PACKAGE__->new->run unless caller;

sub run {
  my ($application) = @_;
  print { $application->{output_fh} }
    "Your work starts here\n";
}

sub new {
  my ($class) = @_;
  my $application = bless {}, $class;
  $application->init;
  $application;
}

sub init {
  my ($application) = @_;
  $application->{output_fh} = \*STDOUT;
}
```

www.TopSage.com

```
sub output_fh {
  my ( $application, $fh ) = @_;

  if ($fh) { $application->{output_fh} = $fh }

  $application->{output_fh};
}

1;
```

现在测试起来方便多了。在测试程序中加载该模块后，构造一个应用程序对象，将输出句柄设为标量引用（见条款54）：

```
use Test::More tests => 3;

my $class = 'MyModulino';
use_ok($class)
  or die 'Bail out! Could not load module!';

my $application = $class->new;
isa_ok( $application, $class );

my $output_string;
open my ($fh), '>:utf8', \$output_string;

$application->output_fh($fh);

$application->run;
like( $output_string, qr/work/ );
```

现在不光能方便地测试MyModulino，还能通过派生它的子类来进行测试。像我们之前独立它的输出文件句柄一样，可以将模块中打印消息的那句话分离出来作为独立的函数，然后在测试脚本中测试该函数的输出：

```
package MyModulino;

use strict;
use warnings;

binmode STDOUT, ':utf8';

__PACKAGE__->new->run unless caller;

sub run {
  my ($application) = @_;
  print { $application->{output_fh} }
    $application->message;
}

# ……中间部分同前

sub message {
  "Your work starts here\n";
}

1;
```

现在不用运行就可以直接测试消息函数的输出了：

```
use Test::More tests => 3;
```

www.TopSage.com

```
my $class = 'MyModulino';
use_ok($class)
  or die 'Bail out! Could not load module!';

my $application = $class->new;
isa_ok( $application, $class );

like( $application->message, qr/work/ );
```

如果想修改输出消息的内容，只需在原来代码上派生出一个非常短小的子类 MyModulino::Icelandic 来，然后重载 message 方法即可，而测试脚本的形式基本保持不变：

```
package MyModulino::Icelandic;

use strict;
use warnings;

use utf8;

use base qw( MyModulino );

__PACKAGE__->new->run unless caller;

sub message {
   "Start pitt byrjar her \n";
}

1;
```

要点

- 将程序写成小模块，可以方便测试和派生子类。
- 将应用程序分成数个片段测试。
- 创建小巧子类对特性进行重载。

条款 92 用虚拟的对象或接口测试

有时要测试的只是代码中的一小部分，而其他不相关的业务逻辑或数据，完全可以用虚拟或替换的方法绕开，从而把注意力集中在被测试代码本身的功能上。

比方说要测试对一组数字求和的程序，实际代码中这组数字从数据库请求中获得：

```
package MySum;

use DBI;

sub sum_values_per_key {
  my ( $class, $dsn, $user, $password, $parameters ) = @_;
  my %results;

  my $dbh =
    DBI->connect( $dsn, $user, $password, $parameters );
  my $sth = $dbh->prepare(
```

www.TopSage.com

大家网 TopSage.com

```
    'select key, calculate(value) from my_table');
  $sth->execute();

  while ( my ($row) = $sth->fetchrow_arrayref() ) {
    $results{ $row->[0] } += $row->[1];
  }

  $sth->finish();
  $dbh->disconnect();
  return \%results;
}

1;
```

当然，我们可以试着创建一个SQLite数据库用于测试（见条款93），但其中用到的calculate()是生产数据库才支持的查询函数，所以这么做行不通。或者我们可以考虑改用依赖注入的方式（见条款89），但有时做起来并不简单，尤其是在无法修改原始代码的情况下。

1. 使用 Test::MockObject 模块

对于MySum中的情形，最简单的办法是顶替掉数据库层。因为代码中直接使用DBI访问数据库，所以我们要创建一个虚拟的DBI对象，以替换掉真实的DBI对象。

为了糊弄Perl让它以为加载的是真实的DBI模块，我们得先“伪造”一个。用fake_module函数指定虚拟对象所属的模块，并加载到内存：

```
use MySum;
use Test::More qw(no_plan);
use Test::Deep;
use Test::MockObject;

my $dbi_mock = Test::MockObject->new;
my $sth_mock = Test::MockObject->new;

$dbi_mock->fake_module( 'DBI',
  connect => sub { $dbi_mock } );
```

接下来就是要模拟所用到的对象方法。在调用DBI->prepare时应该总是返回模拟的查询语句句柄$sth_mock；而每次要断开连接时，总是返回真值表示操作成功：

```
$dbi_mock->set_always( 'prepare', $sth_mock );
$dbi_mock->set_true('disconnect');
```

类似地，我们想要$sth_mock返回测试用数据。这里假设原来代码中的SQL语句语法正确，并能返回所需数据。毕竟，我们要测试的是计算数值求和的功能，而不是数据库及查询语句：

```
$sth_mock->set_true( 'execute', 'finish' );
$sth_mock->set_series(
  'fetchrow_arrayref',
  [ first  => 40 ],
  [ first  => 70 ],
  [ second => 100 ]
);
```

现在可以测试 sum_values_per_key 了。尽管其本身意图是加载真实的 DBI 模块，但我们用 Test::MockObject 构造的虚拟模块取代了它，所以实际上它调用的都是虚拟接口：

```
cmp_deeply(
  MySum->sum_values_per_key(
    'dbi:Oracle:testdb', 'user',
    'password', { RaiseError => 1 }
  ),
  { first => 110, second => 100 },
  'sum values'
);
```

有时，没必要完整虚拟某个对象，简单地重载一两个关键方法就能解决问题。比如，用 Test::MockObject::Extends 替换掉那些方法即可：

```
use Test::More 'no_plan';
use Test::MockObject::Extends;
use DateTime;

my $dt =
  DateTime->new( year => 1979, month => 10, day => 22 );
$dt = Test::MockObject::Extends->new($dt);
$dt->set_always( year => 2009 );

is ( $dt->year, 2009, 'year overridden' );
is( $dt->month, 10, 'month untouched' );
```

2. 通过符号表重新定义方法

有时候 Test::MockObject 和 Test::MockObject::Extends 的使用太过大材小用。必要时修改下符号表也是可以的，不用担心搞乱什么，毕竟只是在测试脚本内改改而已。

如果只是重新定义 DateTime 中 year 方法的返回值，只管在符号表中替换掉函数定义。并且最好把它局限在特定块中，以减少发生意外的可能。这种用法一般比较危险，所以 Perl 会发出 redefine 的警告。当然，我们这是有意为之，关闭 redefine 类型的警告就是了：

```
use Test::More 'no_plan';
use DateTime;

my $dt =
  DateTime->new( year => 1979, month => 10, day => 22 );

{
  no warnings qw/redefine/;
  local *DateTime::year = sub { return 2009 };
  use warnings;
  is( $dt->year, 2009, 'year overridden' );
  is( $dt->month, 10,  'month untouched' );
}
```

现在每次调用 year 方法都会返回 **2009**。

如果要在保留原来函数的同时在其外围增添代码的话，可以定义一个新的函数封装它。重新定义之前，得先将原来函数的定义，即 CODE 类型 typeglob 引用，赋值到临时变量$original，

www.TopSage.com

之后调用时可用&$original这种写法。这里未加圆括号，所以它会以当前的@_作为参数：

```
use Test::More 'no_plan';
use DateTime;
my $dt =
  DateTime->new( year => 1979, month => 10, day => 22 );

{
  no warnings qw/redefine/;
  my $original = *DateTime::year{CODE};

  local *DateTime::year = sub {
    my $year = &$original;
    print STDERR "The real year is $year\n";
    return 2009;
  };

  use warnings;
  is( $dt->year, 2009, 'year overridden' );
  is( $dt->month, 10,  'month untouched' );
  }
```

输出的年份可以证明，重载后的函数内部确实调用了一次原来那个函数：

```
The real year is 1979
ok 1 - year overridden
ok 2 - month untouched
1..2
```

这种方法写起代码来有点累人，又容易出错。可以试试Hook::LexWrap模块，应该会方便很多，而且代码也容易维护。

3. 重载Perl内建函数

除了模块方法及对象外，有时我们还希望替换类似time之类的Perl内建函数。如果不介意完全重载，倒是可以用subs编译指令定义自己的版本，用起来还是和原来一样：

```
my $fixed_time = 1234567890;

use subs qw(time);
sub time {
  return $fixed_time;
}

is( time, $fixed_time, 'frozen in time' );
```

如果需要有更多的选择性，可以重新定义CORE::GLOBAL命名空间内的方法，Perl的内建函数都在该命名空间中定义。不过这里有一点需要注意，重新定义的代码必须放在BEGIN块中：

```
my $fixed_time = 1234567890;

BEGIN {
  *CORE::GLOBAL::time = sub { CORE::time };
}

ok( time > $fixed_time, 'system time' );
```

www.TopSage.com

```
{
  no warnings qw(redefine);
  local *CORE::GLOBAL::time = sub { $fixed_time };
  is( time, $fixed_time, 'frozen in time' );
}
```

这种做法在写测试时非常称手，尤其是需要改变一下代码以防止它在测试时做某些多余的事的时候。比如我们知道 unlink 函数会删除文件，但在测试时我们并不希望它这么干，因为我们要查看原本要删除的内容：

```
BEGIN {
  *CORE::GLOBAL::unlink = sub {
    print "Not unlinking @_";
  };
}
```

现在，所有调用 unlink 的地方都会使用我们改装过的版本。

4. 要点

- Test::MockObject 可用于伪造类以及方法。
- 通过对函数的 typeglob 赋值以重新定义该函数。
- 通过 CORE::GLOBAL 命名空间来重载 Perl 内建函数。

条款 93　用 SQLite 创建测试用数据库

要测试同数据库交互的代码，多少有些挑战。一般的思路是准备一个测试专用数据库，填充一些预备好的测试数据，然后在测试脚本中验证。测试代码非常简单，几乎可以不用构建虚拟对象（见条款 92），而测试环境也和生产环境极为接近。这种做法看起来挺好的，是吧？

事实上，在集成测试时连接实际的数据库是应该的，但在开发过程中需要反复进行单元测试，每次都要重新初始化数据库，如果准备不妥，带来的麻烦可能会比要解决的问题还多。

像 Oracle 或 MySQL 这种大型数据库，建立连接都比较慢，有时甚至会慢到打断开发节奏。特别是测试脚本一多，每个都重复连接和断开数据库的话，这种时间上的消耗更加让人难以接受。多花的那么一两秒钟，足以影响我们的思考连贯性，也会让我们变得不愿频繁运行测试。

此外，同时运行多条测试容易产生对数据库资源的条件竞争。并发测试或手工启动测试的同时，持续集成系统也在定期执行测试的话，很有可能造成数据变化导致其他依赖该数据的测试失败。这种错误是随机出现的，根本无法追踪究竟是程序问题还是测试脚本的问题。

而且万一搞错用户或是弄错连接参数，登录到生产数据库的话，还有可能会对实际运作中的数据产生灾难性的破坏。

1. 本地测试数据库

SQLite 本身并非用 Perl 写成，它是用基于 C 语言的库实现的轻量级本地数据库。援引 SQLite 官方网站（http://www.sqlite.org/）上的定义：“SQLite 作为一套软件库，实现了一种独立自足，

www.TopSage.com

它无需服务器，零配置，并且支持事务处理的SQL数据库引擎。”听起来用作测试会非常完美。

用 SQLite 能省去繁琐的数据库准备和管理，因为它只是本地调用的软件库，所以一切数据库相关的操作都在测试代码内部实现，根本无需建立网络连接，也就没有网络时延的问题。

假设生产代码是要连到 MySQL 的：

```
my $dbh =
  DBI->connect( 'dbi:mysql:my_database', 'user', 'password',
    { other => 'settings' }
  );
```

测试脚本中所需修改的，仅是在 connect 方法中的连接参数。换用 SQLite 驱动：

```
my $dbh =
  DBI->connect( 'dbi:SQLite:database.db', q{}, q{},
    { other => 'settings' }
  );
```

SQLite 会创建（或读取）名为 database.db 的数据库文件。SQLite 兼容绝大多数流行的 SQL 数据库所用的 SQL 语句。就算偶尔碰到不支持的，它也会默然接受，使我们的程序不会因此卡住。

此外，如果不指定数据库文件，就会使用内存数据库。这样，每次连接数据库时，因为数据库还是空的，所以需要在测试时新建数据表，并填充测试用数据：

```
my $dbh =
  DBI->connect( 'dbi:SQLite:', q{}, q{},
  { RaiseError => 1 } );

$dbh->do( 'create table test_table '
    . '( id integer, value varchar )' );
```

SQLite 用于测试十分轻便，当然不光是用在测试程序中，好多开源软件和商业工具都在他们的产品中使用 SQLite 作为后端数据存储。作为轻量级的便携式数据库，又经受了那么多实践的考验，绝对值得我们花些时间仔细研究。

2. 要点

- 不要在测试代码中连接生产数据库。
- 在测试时用 SQLite 替换实际运作的生产数据库。
- 用 SQLite 可以在内存中创建临时的测试用数据库。

条款 94　用 Test::Class 编写结构化测试

虽然 Test::More 提供了各式简单好用的测试函数，但要用于系统化有结构地进行测试便显得力不从心。许多程序员都希望按照当前应用架构专门设计一套组织和维护测试用例的框架。有时候量身定做确实能解决问题，但有时也会落入重复发明轮子的尴尬境地。有 xUnit 背景的人，常常对此颇有微词，甚至一度因为它毫无章法又不够系统而感到失望。

幸而，Test::Class 模块的出现，让习惯 xUnit 风格测试框架的人倍感亲切。Test::Class 是一个完全面向对象的测试框架，后端联合 Test::Builder 等模块，对测试代码进行组织和管理。

www.TopSage.com

和其他 xUnit 风格的测试代码一样，在测试类中创建一系列方法，作为测试架构的基础组件：

```
package MyTest;

use base 'Test::Class';    # 继承测试框架
use Test::More;            # 无需预先告知测试计划
use My::Class::Under::Test;

my %test_data = (
  123 => 'hello',
  456 => 'world',
);

sub connect_to_database : Test(startup) {
  my ($self) = shift;
  $self->{dbh} = connect_to_database;
  $self->{insert_sth} =
    $self->{dbh}
    ->prepare( 'insert into test_table (key, value) '
      . 'values (?, ?)' );
  $self->{delete_sth} =
    $self->{dbh}
    ->prepare('delete from test_table where key = ?');
}

sub disconnect_from_database : Test(shutdown) {
  my ($self) = shift;
  $self->{insert_sth}->finish  if $self->{insert_sth};
  $self->{delete_sth}->finish  if $self->{delete_sth};
  $self->{dbh}->disconnect     if $self->{dbh};
}

sub insert_test_data : Test(setup) {
  my ($self) = shift;
  for my ( $key, $value ) ( each %test_data ) {
    $self->{insert_sth}->execute( $key, $value );
  };
}

sub create_an_object_to_test : Test(setup) {
  my ($self) = shift;
  $self->{object} = My::Class::Under::Test->new;
}

sub clean_up_after_running_a_test_method : Test(teardown)
{
  my ($self) = shift;
  for my ($key) ( keys %test_data ) {
    $self->{delete_sth}->execute($key);
  };
}

sub do_a_single_test : Test {
  my ($self) = shift;
```


www.TopSage.com

```
    is(
      scalar( keys %test_data ),
      $self->{object}->row_count,
      'all data accounted for'
  );
  }

  sub run_a_fixed_number_of_tests : Test(2) {
    my ($self) = shift;
    isa_ok( $self->{object}, 'My::Class::Under::Test' );
    can_ok( $self->{object}, qw(get row_count) );
  }

  sub run_a_dynamic_number_of_tests : Tests {
    my ($self) = shift;
    for my ( $key, $value ) ( each %test_data ) {
      is(
        $test_data{$key}, $self->{object}->get($key),
        "lookup by key $key"
      );
    };
  }
```

可以看到，每个测试函数都标记了相应属性（关于函数属性的概念，本书不作详细介绍，请参阅 **perlsub** 文档说明）。每个属性都有特殊含义，就算不明白工作机理，也应该能从字面上很容易推猜出来：

- Test(startup)

标记 Startup 属性的方法只会运行一次，而且是在其他所有测试方法之前先期运行。一般用来准备测试数据，以备后续各式测试方法取用，而不必每次都重新准备。

- Test(shutdown)

标记 Shutdown 属性的方法也只会运行一次，而且是在所有其他测试方法结束之后运行。一般用于测试程序退出之前做最后的清理工作。

- Test(setup)

标记 Setup 属性的方法会在每个测试方法运行之前自动运行。比方说，有六个普通的测试方法，那么 Setup 方法就会相应地运行六次，每次都在普通测试方法之前运行。一般利用 Setup 方法重建测试用的数据或对象，以便在进行新的测试前使用未经修改过的版本。

- Test(teardown)

标记 Teardown 属性的方法和 Setup 方法相对，在每个测试方法完成之后自动运行，所以可用来清理测试中产生的临时文件或无用数据。

- Test[n]

用以标明实际运行测试函数的方法。可以声明只做一项测试（`Test`），或者多项测试（`Test(n)`），甚至数目不确定的测试（`Tests`）。

现在我们已经把所有测试函数都安置在了对象方法中，除非启动对象调用方法，否则不会自动运行。`Test::Class` 框架提供了一个叫做 `runtests` 的方法，用于启动运行。所以需要我们做

www.TopSage.com

的，仅仅就是调用这个方法而已。

可以另外创建一个 .t 测试文件调用该方法，也可以使用 modulino 技术（见条款 91）在上面的测试模块 MyTest 中写上下面这句：

```
__PACKAGE__->runtests() unless caller;
```

然后就可以直接运行该模块进行测试了：

```
% prove test/MyTest.pm
```

最后需要说明：为便于演示 Test::Class 中的各式挂钩方法，我们在例子中直接连接数据库测试，但我们在写实际的测试代码时，千万注意不要这样直接连到生产数据库。用测试数据库也好，用虚拟接口也好（见条款 92），都比直接连接安全。

要点

- 在 Perl 中可用 Test::Class 模块完成类似 xUnit 风格的测试功能。
- 在测试模块中添加特殊属性的测试方法，辅助完成测试任务。
- 按惯例运行单元测试，或者直接将测试模块当作测试程序运行。

条款 95　项目一开始就准备好测试

现如今，敏捷开发思想和 TDD（Test-Driven Development，测试驱动开发）都非常流行。不管你是 TDD 的忠实拥趸，还是对它不屑一顾，总有许多事实可以证明 TDD 存在的价值。开始编码前不准备好测试，到开发阶段再写，难度要增加不少。

先构建测试，再根据测试要求编写代码使之通过，才是测试驱动开发的真义。所以在 TDD 中，第一步是针对尚未存在的代码编写测试：

```
use Test::More tests => 3;

BEGIN { use_ok('UpperCaser') }

my $uc = UpperCaser->new();
is( $uc->uc('addison clark'),       'ADDISON CLARK' );
is( $uc->uc_first('ella & ginger'), 'Ella & Ginger' );
```

接下来只要运行测试，然后按照 prove 的错误提示（见条款 87），添加或修复相应的程序代码即可。代码不必复杂完备，只要能通过测试就行。第一次运行测试，因为找不到要测试的模块，于是报错：

```
% prove t/uc.t
#     Error: Can't locate UpperCaser.pm in @INC
```

找不到模块是因为它还没有被创建，所以接下来要做的，就是顺着报错信息，先写一个空的模块文件，暂时不必添加任何实质性代码，保存后再次运行测试。这次看到的报错，说找不到 new 这个对象方法：

```
% prove t/uc.t
```

www.TopSage.com

```
t/uc.t .. 1/3 Can't locate object method "new" via ↵
  package "UpperCaser" at t/uc.t line 7.
```

那么现在我们来编写一个空的 new 方法，不返回任何数据。再次运行测试依然失败，报告称，因为没有得到对象，所以无法运行对象方法 uc：

```
% prove t/uc.t
t/uc.t .. 1/3 Can't call method "uc" on an undefined ↵
  value at t/uc.t line 8.
```

于是再修改 new 方法，让它返回尽可能小而简单的对象。再运行测试，报告找不到 uc 方法：

```
% prove t/uc.t
t/uc.t .. 1/3 Can't locate object method "uc" via package ↵
   "UpperCaser" at t/uc.t line 8.
```

加上 uc 方法的代码后，又报错无法定位另一个对象方法 uc_first：

```
% prove t/uc.t
Can't locate object method "uc_first" via package "Upper ↵
  Caser" at t/uc.t line 9.
```

暂时添加一个空的 uc_first 方法，由于它不会作任何处理，所以测试结果和预期不同：

```
% prove t/uc.t
t/uc.t .. 1/3
#   Failed test at t/uc.t line 8.
#          got: undef
#     expected: 'ADDISION CLARK'

#   Failed test at t/uc.t line 9.
#          got: undef
#     expected: 'Ella & Ginger'
# Looks like you failed 2 tests of 3.
```

现在修订 uc 让它完成实际工作，再测试时，就只剩下最后一个方法要修复了：

```
% prove t/uc.t
t/uc.t .. 1/3
#   Failed test at t/uc.t line 9.
#          got: undef
#     expected: 'Ella & Ginger'
# Looks like you failed 1 test of 3.
```

最后完整实现 uc_first 方法功能，再测试，终于全部通过：

```
% prove t/uc.t
t/uc.t .. ok
All tests successful.
Files=1, Tests=3, 0 wallclock secs ( 0.03 usr 0.01 sys ↵
  + 0.02 cusr 0.00 csys = 0.06 CPU)
Result: PASS
```

一路调试修正下来，所有测试均告通过之后，我们最终得到了一段既简单又完整的程序代码：

```
package UpperCaser;

use warnings;
```

www.TopSage.com

```
use strict;

sub new {
  bless {}, shift;
}

sub uc {
  my ( $self, $word ) = @_;
  return uc $word;
}

sub uc_first {
  my ( $self, $word ) = @_;
  return join ' ', map { ucfirst } split /\s/, $word;
}

1;
```

这种形式的开发过程和以往完全不同，它可以保证我们的代码有相应的测试。以往许多人写完代码之后的几天（甚至几分钟），因为觉得太麻烦，会放弃这么做。但是，千万不可逃避！

就算不想照着 TDD 行事，到头来总还是要为代码编写一些自动化测试脚本。写完代码就应该趁热打铁，尽早编写测试，以免事后记忆模糊，留下一摊未经测试的代码混迹其间。而且，能按时达成预定目标也很重要，对于导致测试繁复异常的程序，我们有责任妥善处理化繁为简。因为一般来说，难以测试的代码，同样难以调试，甚至是难以使用。所以测试可以帮我们在开发的早期阶段就预防这类问题发生。

来看一个计数器的例子吧。下面这段代码能从维基百科英文版的首页面上下载数据，提取并打印当下活跃的文章数目：

```
sub print_number_of_wiki_entries {
  my $agent = LWP::UserAgent->new();
  $agent->agent('Mozilla/5.0');
  my $response =
    $agent->get('http://en.wikipedia.org/wiki/Main_Page');
  if ( $response->is_success ) {
    if ( $response->decoded_content =~
      m{>([\d,]*)</a> articles} )
    {
      print
        "There are $1 English articles on Wikipedia\n";
    }
  }
}
```

上面的代码非常简单，这是不是意味着，测试起来也同样简单呢？很抱歉，可能你要失望了，事实上其中多处细节会让编写测试成为挑战。

首先，函数创建了一个自有的 `LWP::UserAgent` 对象，而非从参数列表中取用，所以我们无法用自己的用户代理（user-agent），通过依赖注入的方式测试（见条款 89）。

其次，该函数需要能访问维基百科的网络连接。更糟糕的是，结果直接打印到标准输出，这

www.TopSage.com

就意味着，必须同标准输出数据流打交道（见条款55），才能提取分析数据，判断结果是否正确。

如果先写测试再写代码，就能自然而然地回避绝大多数此类问题。随着测试推进，这类问题会随之显现，当下逐个解决也就轻松自然得多。

当然，这只是一段简单的代码块，针对测试对它进行重构并不困难。最快最明显的改进，是把最后那句打印结论的语句，换成返回字符串：

```
sub get_number_of_wiki_entries {
  my $agent = LWP::UserAgent->new();
  $agent->agent('Mozilla/5.0');
  my $response =
    $agent->get('http://en.wikipedia.org/wiki/Main_Page');
  if ( $response->is_success ) {
    if ( $response->decoded_content =~
      m{>([\d,]*)</a> articles} )
    {
      return "There are $1 English articles on Wikipedia";
    }
  }
}
```

现在可以用字符串比较函数来测试：

```
use Test::More tests => 1;

like(
  ToughToTestRefactored::get_number_of_wiki_entries(),
  qr"There are [\d,]* English articles on Wikipedia",
  'found number of wiki entries'
);
```

将下载数据的部分从数据解析那部分提取出来作为新的函数，原来的程序分两步实现：

```
sub get_wikipedia_main_page {
  my $agent = LWP::UserAgent->new();
  $agent->agent('Mozilla/5.0');
  my $response =
    $agent->get('http://en.wikipedia.org/wiki/Main_Page');
  if ( $response->is_success ) {
    return $response->decoded_content;
  }
  return;
}

sub get_number_of_wiki_entries {
  my $html = shift;
  if ( $html =~ m{>([\d,]*)</a> articles} ) {
    return "There are $1 English articles on Wikipedia";
  }
  return;
}
```

现在可以单独测试下载功能了，利用正则表达式查看抓取来的HTML片段是否符合预期：

```
use Test::More tests => 2;
```

```
my $html = get_wikipedia_main_page();
ok( $html, 'got data back from Wikipedia' );
like(
  get_number_of_wiki_entries($html),
  qr"There are [\d,]* English articles on Wikipedia",
  'found number of wiki entries'
);

my $count = get_number_of_wiki_entries($html);
is( $count, ... );
```

不过目前仍然在用真实的网络连接，一旦离线或者资源不可用，就无法正常运行测试。所以还是用老办法，通过参数传递外部给定的用户代理对象：

```
sub get_wikipedia_main_page {
  my $agent = shift;
  $agent->agent('Mozilla/5.0');
  my $response =
    $agent->get('http://en.wikipedia.org/wiki/Main_Page');
  if ( $response->is_success ) {
    return $response->decoded_content;
  }
  return;
}
```

现在我们来创建一个虚拟对象（见条款 92），替代实际用户代理对象，免去真实连接，直接返回测试数据：

```
use LWP::UserAgent;
use HTTP::Response;
use Test::MockObject::Extends;

my $agent = LWP::UserAgent->new();
$agent = Test::MockObject::Extends->new($agent);
$agent->mock(
  'get',
  sub {
    HTTP::Response->new(
      200, '',
      [ 'Content-Type', 'test/html' ],
      'blah <a>123,456</a> articles'
    );
  }
);

my $html = get_wikipedia_main_page($agent);
ok( $html, 'got data back from Wikipedia' );
like(
  get_number_of_wiki_entries($html),
  qr"There are [\d,]* English articles on Wikipedia",
  'found number of wiki entries'
);
```

现在，我们已经把整个程序分为几个阶段，并分别作了独立测试。要回到单步完成的形式，只需写个封装函数即可：

www.TopSage.com

```
sub print_number_of_wiki_entries {
  print get_number_of_wiki_entries(
    get_wikipedia_main_page( LWP::UserAgent->new() )
  ),
  "\n";
}
```

想想看，这个过程要是放到代码完成几周后，或者添加完一打新特性之后再做，那么编写测试的困难又会提升多少倍。所以，不要制造无谓麻烦，不要和自己过不去：请测试先行。

要点

- 开始编码前，先写测试。
- 初期代码能简则简，通过测试就行，之后再写更多测试，逐步扩增代码内容。
- 重构代码时将程序分离为容易理解的部分，便于分别测试。

条款96　度量测试覆盖率

要追踪哪些代码已经过测试，哪些代码还没编写测试，是一件相当困难的事。从形式上看，测试程序自模块实例化后会调用若干方法，查看结果是否符合预期，但要细数被测试模块的每个函数以确保所有方法都有测试覆盖，这无疑是件繁重而无趣的工作，况且这种地毯式穷举的做法也难免会有疏漏。为了阐明该问题，我们可以想想任意子程序中不同的执行分支，怎么才能保证每条通路都有测试用例覆盖到呢？

举个例子，下面仅有一个函数的模块，其中有条 `if-elsif` 语句将程序流向一分为二：

```
package MyModule;

use warnings;
use strict;

sub do_that_thing {
  my ( $class, $argument ) = @_;
  if ( $argument =~ /x/ ) {
    return 1;
  }
  elsif ( $argument =~ /y/ ) {
    return 2;
  }
  return 3;
}

1;
```

我们按照之前介绍的办法开始测试（见条款95），并自以为所有分支都有测试验证：

```
use Test::More tests => 2;
use MyModule;
is( MyModule->do_that_thing('xyz'), 1, 'got 1 back' );
is( MyModule->do_that_thing('abc'), 3, 'got 3 back' );
```

www.TopSage.com

然后运行 prove（见条款 87），看到所有测试全部通过：

```
% prove -Ilib t/my_module.t
t/my_module.t .. ok
All tests successful.
Files=1, Tests=2, 0 wallclock secs ( 0.03 usr 0.01 ↵
  sys + 0.02 cusr 0.00 csys = 0.06 CPU)
Result: PASS
```

不过，要从这样的结果直观地看出哪个分支被跳过了而没有测试到，几乎是不太可能。上面的测试永远都不会让程序运行 elsif 这块代码，因为根本就没有对应的数据触发它运行。测试结果仅仅针对人为预期的情况，导致它表面看来很完美。这就是问题所在。

1. 让 Devel::Cover 替你盯着

解决这种问题的办法是用 Devel::Cover 模块，它会在测试运行期间记录程序流向，最后给我们一份报告。如果编译环境是用 ExtUtils::Makemaker 辅助工作的话，可以在运行测试前给定环境变量，声明启用 Devel::Cover 模块：

```
% HARNESS_PERL_SWITCHES=-MDevel::Cover make test
```

如果用的是 Module::Build，则指定专门的 testcover 任务即可：

```
% ./Build testcover
```

不管哪种方式，Devel::Cover 都会把收集到的信息保存到 cover_db/ 目录中去。运行 cover 命令，便会把这些信息以总结报告的形式呈现给我们。下面为了符合排印，我们对某些格式做了折行处理：

```
% cover
1..2
Devel::Cover 0.65: Collecting coverage data for branch, ↵
   condition, pod, statement, subroutine and time.
ok 1 - got 1 back
ok 2 - got 3 back
Devel::Cover: Writing coverage database to /Users/jmcadams ↵
  /development/effective-perl-programming/second ↵
  T_edition/esting/code/cover_db/runs/1254690988.30080 ↵
  .46156
---------------------------- ------ ------ ------ ↵
File                           stmt   bran   cond ↵
---------------------------- ------ ------ ------ ↵
lib/MyModule.pm                90.9   75.0   n/a  ↵
t/my_module.t                 100.0    n/a   n/a  ↵
Total                          96.0   75.0   n/a  ↵
---------------------------- ------ ------ ------ ↵
  ------ ------ ------ ------
     sub    pod   time  total
  ------ ------ ------ ------
   100.0    0.0   25.7   84.2
   100.0    n/a   74.3  100.0
   100.0  100.0   91.9
  ------ ------ ------ ------
```

www.TopSage.com

```
Writing HTML output to /Users/Snuffy/MyModule/ ↵
  cover_db/overage.html ...
```

覆盖率测试主要有以下几种类型：

- 语句覆盖率——测试代码中每一条语句。
- 分支覆盖率——测试每一个分支（比如，if-elsif-else 结构中的每一块分支）。
- 条件覆盖率——按所有可能发生的条件判断的每一块代码（比如 $a && $b 中可能出现的两种情况）。
- 函数覆盖率——测试所有的函数。
- Pod 文档——每一个函数都应该有一段 Pod 文档（见条款 82）对应。

报告中对各种类型的代码覆盖率都做了简要概述。Devel::Cover 还一并创建了几个 HTML 报告文件，以具体呈现未经测试的那些代码。cover 命令会在报告输出的最后，给出 HTML 页面文件的路径，我们可以用浏览器打开，看看是哪些代码还没测试到，然后决定从何处开始添加相关测试。

随着项目开发的推进，碰到无需加入覆盖率测试的文件，就会影响测试通过率。比如第三方的模块及其测试文件，通常都无需关注。所以为了避免这种纷扰影响测试结果，我们可以给 Devel::Cover 传递参数指定哪些文件要跳过，哪些要保留：

```
% HARNESS_PERL_SWITCHES=-MDevel::Cover=-ignore,\.t, ↵
  +select,MyModule.pm make test
```

想要简化命令行，可以先设定 PERL5OPT 环境变量：

```
% export \
  PERL5OPT=-MDevel::Cover=-ignore,\.t,+select,.*\.pm
% make test
```

不过，不要一直留着这样的环境变量，否则每次运行测试都要跑一遍 Devel::Cover，而大多时候我们并不关注覆盖率，所以留到最后进行覆盖率测试好了。另外，每次运行测试所产生的数据会逐渐累积，有可能耗尽磁盘空间。要清除之前收集来的数据，可以运行：

```
% cover -delete
```

2. 难以覆盖的代码

如果测试覆盖率达到 100%就完美了，但有时这几乎是不可能的事情。如果你对程序异常的处理极为周到，就很可能会写出下面这样处处设防的代码，以检查所有潜在可能失败的地方：

```
sub very_careful {
  my ($file) = shift;
  open my ($fh), '<', $file
    or die "Could not open $file\n";

  if ( print $fh "Hello there!\n" ) {

  }
  else {
    warn "Could not print!\n";
    unless ( unlink $file ) {
```

www.TopSage.com

TopSage.com

```
        die "Could not unlink $file: $!\n";
      }
    }

    die "Could not close $file: $!\n"
      unless close($fh);
}
```

为达到百分百的测试覆盖率，就必须为每处代码可能的失败进行测试。为 open 失败写一个测试，再为 print 失败另写一个测试，如此以往。构造每个失败的测试比较繁琐，取巧的办法是用虚拟函数替换内建函数（见条款 92），按失败类型测试，这样会方便快捷许多。

你可能会问，不到 100%的覆盖率会不会对我们有影响，是不是还有别的什么办法可以重新组织代码方便测试。通常来讲，代码难以测试，就意味着代码难以维护，也更容易产生潜在的错误。最后我们来考量一下符合经济性的理由：必须付出多少努力，才能保证既定收益？我想，绝大多数时候，99%的覆盖率已经足够，省下的时间应该去做更重要更有意义的事情。

3. 要点

- Devel::Cover 通过多种考量，报告有多少代码已经被测试覆盖。
- 用虚拟替换技术改进测试覆盖率。
- 不必执着于 100%的完美测试覆盖率。

条款 97　把 CPAN Testers 当作 QA 团队

CPAN Testers（http://cpantesters.org/）是一个特别的，由一群志愿者组成的，对上传到 CPAN 的每一个模块进行程序化控制自动运行测试的组织体系。

他们选用各种 Perl 版本，包括非常老旧的版本，以及各式操作系统及配置进行测试。完成测试后，他们会把测试结果上传到官方网站，同时一并发送给模块作者（见条款 68）。有些人设置 CPAN 工具以启用 CPAN::Reporter，以后每次安装新模块，都会上传本地的测试结果。

CPAN Testers 的好处不光是在上传到 CPAN 后对代码的自动测试（见条款 70），它所使用的工具一样也可以在我们内部开发流程中借用。

1. CPAN Testers 设置

有时候 CPAN Testers 给出的报告篇幅很长，大量测试报告蜂涌而至，会令人措手不及，好在 CPAN Testers 允许模块作者对如何收取测试报告进行偏好设置（https://prefs.cpantesters.org/）。用 PAUSE 账号（见条款 70）用户名及密码登录，可以设置默认的通知行为，也可以为每个模块分别设置。

2. 使用开发版本

利用 CPAN Testers 进行测试，不必等到代码完备到能投入生产使用。按照惯例，PAUSE 会把所有版本号码中包含下划线的模块视为开发版本：

```
our $VERSION = '1.001_001';
```

PAUSE 不会索引开发版本，但它仍然会把模块放到你自己的 CPAN 目录中去。而正因为索引文件中没有开发版本，所以用 cpan 工具安装时，也自然不会安装开发版的模块。但 CPAN Testers 会注意到有新的模块上传，它会直接下载，自动测试。一般在上传后几个小时内就会收到来自 CPAN Testers 的测试报告。所以利用这个机制，我们可以把开发中的模块上传到 CPAN，然后通过 CPAN Testers 返回的报告，看看当前模块是否存在兼容性问题。

一般最大的收益来自 CPAN Testers 上广泛的操作系统平台种类。比如说，我们有一段 C 代码需要保证在不同平台和编译工具中，使用正确的数据类型。那就不必自己测试各种可能的情况，只需直接上传，等测试报告就好了。

关于开发版本号的写法，一般是在最后一次发布的版本号后面，跟上下划线及内部的开发迭代号码。比如之前公布的版本为 1.23，那么在下次公布发行之前，应该依次使用 1.23_001、1.23_002 这样的版本号码。等到新版释出，直接改成 1.24 发布，之后继续重复这个过程。

3. 设置自己的 smoke 测试工具

我们可以设置自己的持续集成系统（见条款 98），但借用 CPAN Testers 的工具自己跑自动测试也挺不错（如果不想上传模块到公开的 CPAN 站点，还得设置自己的 CPAN 系统）。CPAN Testers Smoke Tools 页面上有详细介绍（http://wiki.cpantesters.org/wiki/SmokeTools），概括来说无非是选用下面系统中的一种：

- CPAN::Reporter
- CPANPLUS::YACSmoke
- CPAN::YACSmoke
- POE::Component::CPAN::YACSmoke

安装完毕后，可以按照自己的需要修改或配置这些工具。如果不想把测试报告发送到公开的测试报告数据库，可以设为只发送给自己。完整的设置步骤和细节超出本书范畴，但 CPAN Testers 确实非常有用。碰到问题，可以通过他们的邮件列表（testers@cpan.org）咨询。

4. 要点

- 设置 CPAN Testers 通知偏好。
- 通过使用开发版本号，让 CPAN Testers 测试开发中的模块代码。
- 设置自己的 smoke 测试工具。

条款 98　设置持续编译系统

在项目开发中，仅针对当前编写的模块运行测试，得到的结果不免有些狭隘。一个模块的修改，或许会引发其他模块的失败，这种情形并不罕见。所以我们应该间歇运行整个项目的测试套件，当然，最好是将完整的测试套件安置于自动化系统中定期运行。

持续集成系统（continuous integration system）是负责处理代码编译、运行测试等一系列任务的一种框架。小到简单的本地定期运行 prove 命令，大到使用类似 Cruise Control 一样的全功能

www.TopSage.com

系统，在每次版本控制系统中有代码提交时自动运行测试组件。

1. 定期运行 prove

最简单的系统是定期运行测试组件，发送报告到自己的邮箱。只需在定时任务中设置，把 prove 的输出当作邮件内容发送给自己：

```
# 在 crontab 文件中
prove -I/path/to/code/lib /path/to/code/t/*t | \
  sendmail -t me@example.com -s "build results"
```

这么做有个问题。如果代码一直没作修改，那么每次收到的邮件自然也都一模一样。而一旦什么时候编译出错，我们很可能会错过报告邮件，直到很久才会发现。

2. 使用 pre-commit 挂钩

在版本控制系统中加入 pre-commit 挂钩是一种解决办法。在提交任何修改之前，交由该挂钩负责测试，通过测试才能继续提交。在现在各种流行的版本控制系统中，设置这类挂钩程序一般都很简单。

比如用 **Git** 的话，可以在当前项目仓库根目录中添加一个可执行文件 .git/hooks/pre-commit。该程序可以是一个简单的 shell 脚本，每次正式提交前先运行 prove 命令检查：

```
#!/bin/sh
set -e
prove -b t/*t
```

如果用的是 ExtUtils::MakeMaker 这样的编译系统，可以强制 **Git** 在每次提交时都先运行一次完整的测试：

```
#!/bin/sh
set -e
perl Makefile.PL && make test && make realclean
```

Subversion 也可以用类似的挂钩处理，位置在[repo]/hooks/pre-commit，其中 [repo] 是 **Subversion** 仓库的地址。

其他源代码版本控制系统的 pre-commit 挂钩的设置也基本类同。具体实现还请参考各自系统的文档。

随着测试的增加，每次提交都运行一遍完整的测试会叫人不耐烦，特别是频繁提交的话。因此，你必须做好心理预期，为了保留必要的功能测试套件，你和你的团队愿意花上多少时间等待测试结果。

3. 用 Smolder 聚合测试结果

Smolder 是用 Perl 写成的聚合测试结果的工具。我们只需要运行 prove 测试或常规的编译脚本，将结果保存并上传到 **Smolder** 服务器即可，它会负责后续分析和报告。

要尝试运行 smolder，先到 http://sourceforge.net/projects/smolder/ 下载，按照 INSTALL 文件中的说明安装并设置，启动该服务器。

都准备好之后，用 prove 运行测试，将归档的测试结果发送到 **Smolder** 服务器上：

www.TopSage.com

```
% prove --archive=/tmp/build.tgz \
  /path/to/tests/t/*t && smolder_smoke_signal \
  --server smolder.example.com --username username \
  --password password --file /tmp/build.tgz \
  --project MyProject
```

服务器会以漂亮的基于 Web 页面的方式展示测试历史，而且仅当测试出现失败时才通知你。

我们也可以利用 SmokeRunner::Multi 模块监视代码仓库的特定分支，一旦发现有所变更，立即自动运行测试，并把测试结果提交到 **Smolder** 服务器上。这样的做法，和流行的持续编译系统 Cruise Control 的做法如出一辙。

以上两个例子都是简单地在本地磁盘上运行测试。如果正好修改某些文件，便有可能导致这些测试失败。直到这个阶段的开发完成，否则聚合测试的结果就会一直处于测试失败的状态。当然，我们可以边敲代码，边从版本控制系统自动检出之前的版本跑测试，不过在此之前，先来花些时间看看持续编译系统是如何工作的。

4. Cruise Control

Cruise Control（http://cruisecontrol.sourceforge.net/）是一个用 Java 编写的持续集成并运行测试的应用框架。只需在我们的 Perl 项目中稍加改动，便可立即运行 **Cruise Control** 为我们服务。

- 用 Ant 编译

为了在我们的项目中使用 **Cruise Control**，最好编译和测试都使用 **Apache Ant** 来完成。如果用的是 Module::Build 编译脚本，那么 ant 对应的 build.xml 文件应该大体如下：

```
<project name="myproject" default="all">
  <macrodef name="module.build">
    <attribute name="action" default="build" />
    <sequential>
      <exec executable="/usr/bin/perl"
        failonerror="true">
        <arg value="Build" />
        <arg value="@{action}" />
        <env key='PERL5LIB' path="lib:${env.PERL5LIB}" />
        <env key="PERL_TEST_HARNESS_DUMP_TAP"
          path="/project_dir/target/test-results/"/>
      </exec>
    </sequential>
  </macrodef>

  <target name="all"
    depends="clean, configure, build, test"/>

  <target name="clean">
    <module.build action="clean"/>
  </target>
  <target name="configure">
    <exec executable="/usr/bin/perl" failonerror="true">
      <arg value="Build.PL"/>
        <env key='PERL5LIB' path="lib:${env.PERL5LIB}" />
    </exec>
```

www.TopSage.com

```
  </target>

  <target name="build" depends="configure">
    <module.build action="build"/>
  </target>

  <target name="test" depends="build">
    <module.build action="test"/>
  </target>
</project>
```

乍看之下，这个文件貌似复杂异常，但其实它还是非常简单的。它所做的，不过是定义了一个名为 **myproject** 的项目，然后封装了若干编译任务，诸如清理文件，运行 Build.PL，构建项目代码，测试项目，等等。项目叫什么名字由你而定，只要你和你的团队觉得没问题就行。

另外还要修改 PERL_TEST_HARNESS_DUMP_TAP 环境变量的取值，测试台将根据此变量决定存放测试输出副本的位置，即 **Cruise Control** 最终收集信息的目录。它可以在独立于项目目录以外的地方，当然也可以保留在项目目录中，但一定要记得让版本控制系统忽略该目录。

写完 build.xml 文件之后，便可以用 ant 对项目配置、编译、测试以及清理了：

```
% ant configure
% ant build
% ant test
% ant clean
```

这里的 build.xml 文件借鉴了 TAP::Formatter::TeamCity 模块文档中的 build.xml 文件，但它更为灵活健壮，值得学习。

- 格式化输出

改用 ant 编译和测试项目代码确实是一个比较大的动作，但不光如此，为了结合使用 **Cruise Control**，还得做其他方面的改动。到目前为止，**Cruise Control** 可以从测试结果判断整体上是成功还是失败，但这还不够，我们的目的是要让它对每条测试都给出解释，所以必须将原本是 TAP 格式的测试结果，转为 JUnit 输出的 XML 格式。

若是恰好在用 Module::Build，重新格式化输出无比简单：只需要继承 Module::Build 写个子类，重载方法，然后让我们的 Build.PL 文件加载使用这个类即可。

第一步，构建重载 tap_harness_args 方法的类，然后选用 TAP::Formatter::JUnit 作为格式化工具：

```
package MyModuleBuild;
use parent 'Module::Build';

sub tap_harness_args {
  return { formatter_class => 'TAP::Formatter::JUnit' };
}

1;
```

然后修改 Build.PL 文件，使之改用我们自己的 MyModuleBuild 模块：

```
use MyModuleBuild;
```

www.TopSage.com

大家网 TopSage.com

```
my $build = MyModuleBuild->new(
  module_name => 'MyModule',
  dist_version => 1,
);

$build->create_build_script;
```

现在运行 ant test，我们会看到屏幕上的输出成了 JUnit XML 格式。到环境变量 PERL_TEST_HARNESS_DUMP_TAP 定义的目录中查看，也会发现若干 XML 文件。

但若用的不是 Module::Build，改起来就不那么简单了。倒是可以在测试任务中直接指定使用 prove：

```
<target name="test" depends="compile">
  <exec executable="/usr/bin/prove"
    failonerror="true">
    <arg value="--formatter=TAP::Formatter::JUnit"/>
    <arg value="t/"/>
    <env key="PERL_TEST_HARNESS_DUMP_TAP"
      path="/project_dir/target/test-results/"/>
  </exec>
</target>
```

- 设置 **Cruise Control**

现在我们的项目已经改成使用 ant 编译，并以 JUnit 的方式输出 XML 格式的测试结果，都准备好了，是时候来使用 **Cruise Control** 了。请先下载并安装这个应用框架。可以在它的项目主页中找到下载链接。

完成下载 **Cruise Control** 之后，解压缩该文件，可以将得到的目录移至任何我们喜欢的位置，只要能轻易调用其中的可执行文件就行。在默认情形下，**Cruise Control** 会把该目录用作存储数据，所以确保有足够的磁盘空间，以备日后数据增长所需。

编辑 **Cruise Control** 目录中的 config.xml 文件进行配置，去掉示例项目，为自己的项目添加条目：

```
<cruisecontrol>
  <project name="myproject">
    <listeners>
      <currentbuildstatuslistener
        file="/cc_dir/projects/myproject/status.txt"/>
    </listeners>
    <bootstrappers>
      <antbootstrapper anthome="apache-ant-1.7.0"
        buildfile="/project_dir/build.xml"
        target="clean"/>
    </bootstrappers>
    <modificationset quietperiod="0">
      <filesystem folder="/project_dir/"/>
    </modificationset>
    <schedule interval="300">
    <ant anthome="apache-ant-1.7.0"
      antWorkingDir="/project_dir/"
```

```
      buildfile="/project_dir/build.xml"/>
    </schedule>
    <log>
      <merge dir="/project_dir/test-results/t/"/>
    </log>
  </project>
/cruisecontrol>
```

我们需要给项目命名，以便在 **Cruise Control** 的使用界面中有所区分。把 /cc_dir 改为我们自己的 **Cruise Control** 所在路径。每个 myproject 都应该替换为我们实际项目的名称，而每个 project_dir 也都应该替换为我们的 Perl 项目源代码所在目录。

完成配置 `config.xml` 文件后，启动 **Cruise Control**：

```
% ./cruisecontrol.sh
```

现在打开浏览器，访问 http://localhost:8080 应该会看到持续编译系统如期运行。目前这还只是些基本的配置，为了适应我们自己的开发环境，还有许多地方可以定制 **Cruise Control** 的行为。

5. 其他方案

另外，有许多其他集成编译系统也可以为我们的开发服务。`Test::AutoBuild` 是用 Perl 写的系统，自然可以用于 Perl 项目的开发。**Hudson** 是另一个流行的 Java 服务器，可以理解 `Test::Harness` 输出的测试结果。

你的公司可能已经规定了应使用何种集成系统，又或者完全没有限制，只是在开发过程中随需而定。但这都不重要，用什么系统都行得通，关键是有这样一套集成测试体系，用以保障软件开发的效率和质量。

6. 要点

- 用持续集成系统自动运行测试。
- 用功能完善的支持 Perl 的持续编译系统。
- 不管采用哪种持续集成系统，都远胜过没有使用集成系统。

www.TopSage.com

第 11 章 警告信息 11

Perl 可以帮我们定位可疑代码并给出相关提示。从其他语言中学到的大部分调试技能在 Perl 里面也同样适用，所以碰到程序输出整页整页的警告信息时，完全可以利用先前掌握的技能开始调试。

打开警告信息的方式有许多种，老式的 Perl 是在命令行启用 -w 开关：

```
% perl -w program.pl
```

这等效于放在 shebang 行里面：

```
#!perl -w
```

而现在的 Perl 程序，一般都是在代码里直接用 warnings 编译指令（见条款 99）：

```
use warnings;
```

处理警告信息是有技巧的，不但需要弄清楚 Perl 给我们的警告是什么意思，还要弄清楚产生警告的缘由和背景。有些问题可以通过阅读源码来定位，但也有些问题是需要一定时机才能重现。

另外 Perl 还有个很棒的特性，叫做**污染检查**（taint checking），它能跟踪进入程序的外部变量，并防止程序直接将其传递给外部程序。这个特性可以防止程序接受外部恶意数据，以免受到“污染”。

当然，对于这些警告信息其实也不必过于敏感，它们不过是提示可能存在的问题。所以并不是说，出现了警告就一定要不分轻重全部解决。Perl 有一个在指定作用域内临时关闭警告的特性，如果我们对某项警告了如指掌，知道它肯定不会有什么问题，又不想兴师动众上下翻改，就可以利用这个特性临时关闭警告输出。

条款 99 启用警告功能定位可疑代码

不论是在编译还是运行程序，一旦发现可疑代码，Perl 都会发出警告。警告是非常有价值的开发调试工具，我们不能忽略。如果以前写代码一直不够严谨，那么经过几个月 Perl 不厌其烦地汇报错误发出警告，我们就能渐渐养成好习惯，写出警告越来越少的代码。尽管如此，最好还是保持开启 Perl 警告功能，大多数时候，有总比没有好。

为了兼容以前写好的脚本，Perl 是默认关闭警告功能的，但我们可以通过多种方式打开。从

www.TopSage.com

Perl 5.6 开始，我们可以针对单个文件打开警告功能，这也是我们极力推荐的方式：

```
use warnings; # 将这行放在程序文件的开头
```

如果项目文件没有 use warnings 这行，我们仍可以在运行时加上 -w 开关打开全部警告功能，像这样：

```
% perl -w myscript
```

在程序的第一行（也称作 shebang 行）加上-w 开关也能启用警告功能：

```
#!/usr/bin/perl -w
```

另外，我们也可以有选择地在部分代码块中启用或者禁用警告功能（见条款 100）。

1. 编译时警告

在编译阶段，Perl 就能发现某些可疑问题。请看下面这段简单程序，其中有一条加法运算表达式：

```
use warnings;
$foo = 1;
1 + 2;
```

我们已通过 -w 开关启用警告功能，当使用 -c 开关检查程序语法时，perl 就会告诉我们那行加法表达式其实基本无用，因为其运算结果根本没派上任何用场，而且这段代码中的变量也只出现了一次，所以这很可能是个错误：

```
% perl -cw program.pl
Useless use of a constant in void context
Name "main::foo" used only once: possible typo
/home/snuffy/program.pl syntax OK
```

2. 运行时警告

运行时的警告来自于源代码中那些 perl 无法在编译时察觉的错误，通常也是最麻烦的。假设我们要打印某个散列的键值对：

```
use warnings;

foreach my $cat ( keys %microchips ) {
  print "$cat --> $microchips{$cat}\n";
}
```

你能从源代码中看到可能导致警告信息的问题吗？事实上，perl 也看不出来，只有到它运行程序遍历散列时才会发现有个未定义值：

```
my %microchips = (
  'Mimi' => 123,
  'Buster' => undef,
  'Roscoe' => 345,
);
```

perl 运行到 Buster 这个键的时候，会打印出它未定义的值，同时发出一个警告信息：

```
Use of uninitialized value $microchips{"Buster"}↵
  in concatenation (.) or string
```

www.TopSage.com

很麻烦吧，但这条警告已经比以前的好很多了，以前不会明确给出哪个键值未定义，但从Perl 5.10开始，警告信息会清晰告诉我们是哪里使用了未定义值，在本例中就是$microchips{"Buster"}。

由于我们很难通过阅读源代码发现运行时才发出的警告，捕获这些问题的最好方法就是建立一套严谨的测试集，使用非常规的数据和异常情况来覆盖各种状况（见条款96）。

3. 通过 diagnostics 获取更详尽信息

如果仅仅只是打开警告功能，就很有可能碰到某条看不明白的警告信息。哪怕你已经使用Perl很长时间，都有可能碰到从未见过的警告信息。为了更好地了解警告信息的含义，我们可以查看**perldiag**文档获取各种出错信息的详细解释。大部分解释都会提供引发这些问题的常见原因。

除了深入阅读文档外，当然还有更简单的方法用以获取详细警告信息。我们可以使用diagnostics编译指令将简短的出错警告换成详细的出错信息：

```
# use warnings;
use diagnostics;
$foo = 1;
```

现在这个程序的警告输出就更详细具体了：

```
% perl -c program.pl
Name "main::foo" used only once: possible typo ...
    (W once) Typographical errors often show up as unique
    variable names. If you had a good reason for having a
    unique name, then just mention it again somehow to
    suppress the message. The our declaration is provided
    for this purpose.

    NOTE: This warning detects symbols that have been used
    only once so $c, @c, %c, *c, &c, sub c{}, c(), and c
    (the filehandle or format) are considered the same; if
    a program uses $c only once but also uses any of the
    others it will not trigger this warning.
```

4. 生产环境下的警告

在生产环境中启用警告是颇具争议的，所以提到这个时请做好面对其他不同声音的心理准备。这与面对代码中括号的使用风格是制表符还是空格，以及对可口可乐和百事可乐孰优孰劣这类问题一样，你完全可以根据自己的意愿决定。

运行时警告会对程序的性能产生一点影响，此外，将意外或不实的警告信息呈现给最终用户也绝非良策。所以通常情况下，警告信息应面向开发者，而不是用户。我们应该在发布应用程序代码前先关闭警告功能。由于我们可以选择性地启用警告功能并检测是否启用（见条款100），所以我们随时都可以在必要时重启警告，以便调试程序。

不同Perl版本间警告信息的变化也是个大问题。除了产生新的警告信息，之前无关痛痒的警告信息可能会变成严重警告。你一定也不想在部署应用程序时承担这样的风险吧？

那些倾向于“无论合适与否都要打开警告功能”的人肯定没有在周末凌晨被叫起来工作过，

www.TopSage.com

然后发现仅仅因为某项 perl 升级导致以前没有警告且运行良好的应用突然开始不断产生新的警告信息的经历。这些不断产生的警告信息会导致日志耗尽硬盘空间，而实际上这个程序运行十分正常，真正的问题只在于没人监控这些无关痛痒且消耗硬盘空间的警告信息。

如果你还是想留着警告信息输出，那也没关系，只需明白警告功能不但随时都可以打开，反过来也一样——要关随时都可以关。而在企业级应用中，可以采用类似 Log::Log4perl 这样的模块（见条款113），对经过良好测试的关键业务代码进行消息记录，另外配合一套适合的测试脚本，在部署前捕获并解决所有潜在的警告信息。

5. 要点

- 用 perl -cw 开启编译时警告。
- 用 use warnings 针对单个文件启用警告。
- 用 use diagnostics 换成更详细的警告信息。

条款 100　利用词法作用域选择性启用或关闭警告

虽然警告只表示代码存在问题的可能，并不足以严重到让程序中止运行，但还是应该对警告信息多加留心。

比如下面的代码，对于没有初始化值的 $number 变量我们可能无所谓。因为在接下来的计算表达式中 Perl 会将其转换成数字 0，所以大抵没什么问题：

```
my $number;
my $value = $number + 2;
```

但如果启用 warnings 的话，Perl 就会对此抱怨：

```
Use of uninitialized value $number in addition (+)
```

像这种情况，我们实在没必要操心 $number 变量一开始是否有值，因为无论如何，后面都会正确转换。这个例子还算简单，但有时候确实会碰到一些无法忽略的警告，必须逐一排除，这就让人头疼了。解决办法是临时关闭警告功能。

warnings 编译指令是满足词法作用域的，所以我们可以将不需要警告的代码块包起来，取消导入 warnings 编译指令：

```
{
  no warnings;

  my $number;
  my $value = $number + 2;
}
```

上面这段代码仍然有个小问题，它把所有警告信息都关闭了，因此很可能会错过一些重要的警告信息。

warnings 编译指令允许将警告信息分类，所以我们可以有选择的启用或禁用不同类型的信息，请阅读 warnings 的文档以获取完整列表。而在本例中，我们可以只关闭那个 uninitialized

类型的警告：

```
{
  no warnings 'uninitialized';

  my $number;
  my $value = $number + 2;
}
```

1. 提升某些警告的严重级别

对于某些非常讨厌的问题，我们可能想要将它们从仅仅只是警告提升到严重错误的级别。比如当 perl 没法将字符串完整转换成数字时，通常会得到一个警告信息，代码如下：

```
use warnings;
my $sum = '123buster' + 5;
print "The sum is $sum\n";
```

虽然发出了警告，但 perl 仍然会继续运行，于是得到的结果是：

```
Argument "123buster" isn't numeric in addition (+)
The sum is 128
```

如果不希望得出这样的错误结果，我们可以人为提升警告的严重级别，设定它为致命错误(fatal)：

```
use warnings FATAL => 'numeric';
my $sum = '123buster' + 5;
print "The sum is $sum\n";
```

好了，现在一旦发出警告，程序就会中止运行，因为计算非数值参数的加法已经成了一项严重错误：

```
Argument "123buster" isn't numeric in addition (+)
```

2. 在模块中使用预定义的警告类型

我们可以在词法作用域内自定义警告信息。除了简单地用 warn 发出警告信息外，我们还可以用来自 warnings 编译指令的 warnif 函数发出警告信息，并能自定义警告信息的类型：

```
sub create {
  warnings::warnif( 'deprecated',
    'create is deprecated, use new instead' );
  ...;
}
```

如果正在使用 carp 模块（见条款 102），可以在触发 carp 方法前先测试一下某个警告类型是否启用：

```
use Carp;

sub create {
  carp('create is deprecated, use new instead')
    if warnings::enabled('deprecated');
    ...;
}
```

www.TopSage.com

尽管 warnif 也可以响应严重警告，但在调用 carp 方法前是没法检测某个特定警告类型是否属于严重级别。

3. 创建自己的警告类型

关于词法作用域警告的内容远不止上面提到的这些，如果你发现预定义的警告类型不能满足需求，那就可以像预定义警告类型一样自己定义，然后有选择地开启或关闭。

4. 要点

- 用 no warnings 在某个词法作用域中临时关闭警告。
- 如果想忽略某一类型的警告，可以单独禁用它。
- 如有必要，可以将某些警告提升至严重错误级别。

条款 101　用 die 抛出异常

有些人自老式的 C 程序风格中继承了一些习惯，他们会在每个函数结尾返回一个代表执行成功与否的代码。比如，为了从子程序中返回一个错误，他们会返回 undef：

```
sub do_work {
  my $task = shift;

  if ( $task < 0 ) {
    return;  # 有错误产生，返回 undef
  }

  ...;
}
```

虽然可以这么做，但随后我们不得不花更多力气去验证子程序是否正常工作：

```
my $value = do_work($task);

if ( defined $value ) {
  print "It worked and I got [$value]!\n";
}
else {
  print "Something went wrong, but I don't know what!\n";
}
```

如果发生错误，虽然可以从返回值知道有错发生，但却没法知道究竟错在何处。碰到这种问题，我们可以直接使用 die，让它抛出异常，给出更有意义的出错信息：

```
sub do_work {
  my $task = shift;
  if ( $task < 0 ) {
    die("Task [$task] should be greater than zero");
  }
  ...;
}
```

www.TopSage.com

瞧，改动虽然不大，却给我们带来了额外好处：一旦运行到这里，程序就会停止，以迫使我们处理这个问题。我们还可以用 eval 函数对它进行捕获。如果 do_work 中的 die 运行了，那么 eval 会捕获到它并将 die 输出的信息保存到特殊变量 $@ 中去：

```
my $value = eval { do_work(@tasks) };
if ($@) {
  # 处理错误
}
```

在 eval 之后，我们需要马上检查特殊变量 $@，以免其他异常覆盖它的值（见条款 103）。

eval-if 这样的习语和其他语言中的 try-catch 差不多，当然我们也可以在 Perl 中使用 try-catch（见条款 103）。

而一种更好的方式是：把一个对象引用作为 die 参数抛出，该对象引用可以是一个代表出错内容的对象。比如，借助出错记录模块创建出错信息对象：

```
sub do_work {
  my $task = shift;

  if ( $task < 0 ) {
    die MyErrors->fatal("Task should be greater than 0");
  }
  ...;
}
```

如果 eval 中的语句运行失败，我们就可以从 $@ 中取出出错信息对象，然后像操作其他对象一样操作错误信息，因此更加灵活：

```
my $value = eval { do_work(@tasks) };

if ( my $error = $@ and ref $error ) {
  print "I found an error at level ", $error->level, "\n";
  print "The message was ", $error->message, "\n";
}
```

并不是所有情况都必须调用 die，如果我们用 autodie（见条款 27）处理 Perl 内置函数的出错信息，就不需要显式调用 die 了。我们还可以用 Try::Tiny 模块（见条款 103）更好地进行异常处理。

要点

- 用 die 产生异常。
- 给 die 一个引用或着对象作为参数，传递结构化的出错信息。
- 用 eval 捕获异常。

条款 102　用 Carp 来获得栈跟踪信息

如果在代码调用 die 或发出警告信息时，Perl 能尽量给出关于错误根源的详细信息该有多好，特别是有关调用栈的跟踪信息。遗憾的是，Perl 内置的 die 和 warn 函数都没法提供这些信息。

www.TopSage.com

下面列出一些调用 randomly_fail 的子程序，其中某些子程序还会相互调用。在运行这段程序之前，我们并不知道何处会出错：

```
sub randomly_fail {
  die 'ouch!' if int( rand(100) ) % 10 == 0;
}

sub caller_one {
  randomly_fail();
}

sub caller_two {
  caller_one();
  randomly_fail();
}

sub caller_three() {
  caller_two();
  randomly_fail();
}

while (1) {
  caller_three();
}
```

运行该程序，我们看到它（正如预期一样）发生错误，只不过对于哪里出的错，为何会出错，一概无从知晓：

```
% perl randomly_fail.pl
ouch! at die.pl line 2.
```

这样的出错信息包含的信息量十分有限。我们当然知道它是在代码第 2 行调用 die 函数时退出的，而 die 函数在源代码中所处的位置，我们本来就知道，自不用多说。而 warn 函数也存在同样的问题。

1. 用 Carp 模块获取更多信息

Carp 属于核心模块，它能给我们比 die 或 warn 函数更多有用的出错信息。我们可以分别用 croak 和 carp 函数替换掉 warn 和 die 函数，这样就能从函数调用者的角度得到完整的栈跟踪信息。这时候的出错信息不仅会告诉我们代码是在哪里停止的，它还会告诉我们更深层的内容，即调用子程序的逐层路径：

```
package Bar;
use Carp;
sub fail { croak "Ribbit!" }

package main;

Bar::fail();
```

尽管这个程序是在第 3 行停止运行的，但第 7 行才是因调用 Bar::fail 而产生问题的地方，这也正是 croak 函数报告之处：

www.TopSage.com

大家网 TopSage.com

```
Ribbit! at carp.pl line 7
```

当调用者与异常发生不在同一个包内时，我们得到的就是如上信息。Carp 模块会在调用栈中搜索，直到找到包发生变化的地方。

如果在调用栈内的包中 Carp模块没有发现变化，则会给出完整的栈跟踪信息。现在我们可以把 die 改为 croak，改进第一个代码范例了：

```
use Carp;

sub randomly_fail {
  croak 'ouch' if int( rand(100) ) % 10 == 0;
}

sub caller_one {
 c randomly_fail();
}

sub caller_two {
  caller_one();
  randomly_fail();
}

sub caller_three() {
  caller_two();
  randomly_fail();
}

while (1) {
  caller_three();
}
```

这段程序每运行一次都有可能给出不同的出错调用路径：

```
% perl carp.pl
ouch at carp.pl line 4
  main::randomly_fail() called at carp.pl line 8
  main::caller_one() called at carp.pl line 12
  main::caller_two() called at carp.pl line 17
  main::caller_three() called at carp.pl line 22

% perl carp.pl
ouch at carp.pl line 4
  main::randomly_fail() called at carp.pl line 18
  main::caller_three() called at carp.pl line 22
```

如果这是你所关心的正在维护中的程序，那么现在用这个方法就能跟踪各种出错情况了。

2. 总是给出栈回溯信息的 confess

我们不一定非得在同一个包中才能获得完整的栈回溯信息。如果我们将 croak 换成 confess，那现在所有的信息都能得到了：

```
package Bar;
use Carp;
```

```
sub fail { confess "Ribbit!" }

package main;

bar::fail();
```

现在，尽管程序是在和调用者不同的包中调用 die 退出的，我们还是能获得栈的回溯信息：

```
Ribbit! at carp.pl line 3
  bar::fail() called at carp.pl line 7
```

这个特性使得 confess 成了一个很好的调试工具，但在生产环境中我们恐怕不会用到。如果不是为了调试，用 croak 就够了。通过将变量$Carp::Verbose 的值设为真，我们可以把 croak 变成和 confess 一样的功能，也就是说，我们可以临时定义它的实际功效：

```
{
  local $Carp::Verbose = 1;

  Bar::fail();
}
```

如果想把所有 croak 变得和 confess 一样，我们只需改变 Carp 模块的导入参数即可：

```
use Carp 'verbose';
```

这样的话，虽然代码中仍然用的是 croak，但当临时需要调试程序时，便可以开启并获得完整的栈跟踪信息。

3. 要点

- 用 Carp 模块获得导致代码出错或产生警告的相关信息。
- 用 confess 获取完整的栈回溯信息。

条款 103　正确处理异常

Perl 没有内置的异常处理机制，所以我们需要自己用 eval 和 if 仿造一个（见条款 101）：

```
my $value = eval { die "throw an error"; };
if ($@) {
  warn "I caught an error: $@";
}
```

然而，这里有个大问题，因为我们可以将任何值作为参数传递给 die 函数，包括假值。比如在下面这段程序中，$@里面是假值，所以 if 语句无法捕获：

```
my $value = eval { die ''; };
if ($@) {
  warn "I should have caught an error: $@";
}
```

1. 使用 Try::Tiny 模块就对了

处理前面说到的这种情况，最合适方法的就是 Try::Tiny 模块中展示的方法。虽然 CPAN 上有很多其他异常处理模块，但它最为轻巧，也没有问题多多的依赖关系。如果我们仍想用标准

www.TopSage.com

的 Perl 完成和这个模块一样的工作，只需要照着该模块中的例子做就对了。

首先，我们需要正确处理特殊变量 $@。这比很多人想象的要更困难更微妙些。我们需要避免两个问题：不能修改上层 eval 的 $@ 值，也不能让自己嵌套的某个 eval 改变之前的 $@ 值，哪怕这个被嵌套的 eval 隐藏在某个函数调用之后。在使用 $@ 变量前我们先要对它进行局部化。一旦运行完 eval 语句，应该立即保存 $@ 变量到自己的变量中，以确保不会产生意料之外的内容变化。下面的代码片段来自 Try::Tiny 模块，展示了解决办法的主要部分：

```
my ( $error, $failed );
{
  local $@; # 防止改变上层的相同变量
  $failed = not eval {
    die "throw an error";
    return 1;
  };
  $error = $@; # 防止来自下方对 $@ 变量的修改
}
if ($failed) {
  warn "Caught an error: $error";
}
```

Try::Tiny 会帮我们处理这样的麻烦问题，并提供了一般的 try/catch 语法：

```
use Try::Tiny;

try {
  die "throw an error";
}
catch {
  warn "Caught an error: $_";
};
```

注意 catch 代码块是以分号结尾的。它和 eval 一样，是个 Perl 表达式。另外，它会将出错信息保存在变量 $_ 中而不是 $@ 中（这也是为什么需要捕获异常的代码放在前面的 try 中，而抛出异常放在后面的 catch 中的原因）。

当然，我们也可以传给 die 一个引用（见条款 101），然后使用 given-when 语法（见条款 24）过滤出错信息：

```
use Try::Tiny;
use 5.010;

try {
  die MyError->new(...);
}
catch {
  when ( $_->type eq 'IO' ) { ... }
  when ( $_->type eq 'Fatal' ) { ... }
  default { ... };
};
```

如果想忽略出错信息，只需要去掉 catch 块就行，很简单吧：

```
use Try::Tiny;
```

www.TopSage.com

```
try {
  die "throw an error";
};
```

2. Try::Tiny 陷阱

有时候看起来像是在使用 Try::Tiny 模块，但实际上并不是。下面这个例子中，我们忘记加载 Try::Tiny 模块了，但从语法上来说它在 Perl 里面仍然是合法的，所以 Perl 并不会抱怨，即便是在严格约束条件下（见条款 3）。

```
use strict;

try {
  die "throw an error"; # 无论如何都会运行 die
}
catch {
  warn "Caught an error: $_";
};
```

另外，由于 try-catch 中的代码块也是真正的 Perl 子程序，所以我们不应该在里面使用 return 函数，否则它会被搞懵的：

```
try {
  die "throw an error";
  return; # 错了!

}
catch {
  warn "Caught an error: $_";
};
```

3. 要点

- 用 eval 捕获错误，通过 $@ 检查出错信息。
- 处理 $@ 需慎重，Try::Tiny 模块可以正确完成这项工作。

条款 104　通过污染检查跟踪危险数据

污染检查是 Perl 的一个运行时特性，可以用它跟踪数据在程序中的动向。如果数据来自用户输入或程序外部传入，那么 Perl 会认为这些数据可能存在安全隐患，比如命令行参数值、环境变量、文件内容或流式输入等。在污染检查模式下，这些数据是无法直接传递给外部程序的，如果一定要这么做，Perl 会强制停止程序运行。

我们可以通过打开 -T 开关启用污染检查模式：

```
% perl -T program.pl
```

当然，也可以在 shebang 这行启用：

```
#!perl -T
```

在某些情况下，比如运行在 setuid 的特殊权限模式下，perl 会自动打开污染检查模式。

www.TopSage.com

如果在程序内部启用了-T，但在命令行上直接运行该程序的话，perl 就会报错：

```
% perl program.pl
"-T" is on the #! line, it must also be used on the ↵
   command line
```

为了说明污染检查模式是如何工作的，我们假设有这样一个程序，它从用户那里获取一个 glob 的匹配模式，然后将这个模式传递给外部 grep 命令：

```
#!perl -T
print "Enter a grep pattern: ";
chomp( my $pattern = <STDIN> );
print `grep $pattern *`;
```

如果在污染检查模式下运行该程序，perl 会在程序开始实际工作前停止，提示如下：

```
Insecure dependency in `` while running with -T switch
```

上面程序中的 $pattern 变量值是从标准输入直接获得的，而它的值可以是任意形式的数据，因此，直接将用户输入传递给 shell 程序运行，显然是有问题的。假设用户输入一些包含特殊字符的内容，比如利用表示 shell 当中一条命令结束的；字符，就可以同时运行多条 shell 命令。万一有人攻击，输入下面的内容就很危险了：

```
% perl program.pl
enter pattern: ; rm *
```

而在污染检查模式下，perl 会强迫我们在将数据传递给外部进程前先做净化处理。要使它们变为未污染的数据只能使用带捕获变量的正则表达式，且最好是尽量让构建的模式能明确地只匹配我们所允许的形式（Mark Jason Dominus[①]称之为“Prussian Approach”）。举例来说，作为 grep 命令的模式参数，如果我们想要别人输入的内容只能包含字母、点号（.）和量词星号（*），那我们的正则就应该只能匹配这些字符：

```
#!perl -T
my $tainted_data = <STDIN>;

my $untainted_data = do {
  if ( $tainted_data =~ m/^([a-z.*])$/i ) {
    $1;
  }
  else {
    die "The pattern can only contain letters, ., or *!\n";
  }
};

print `grep '$untainted_data' *`;
```

在这个例子中，如果输入内容和我们期望的不同，那么程序会失败退出。如果这个正则匹配模式漏掉某些我们想要的东西，没关系，我们可以再作适当修改，至少我们不会有安全上的问题，

① Mark Jason Dominus 是一位 Perl 编程领域的领跑者，曾担任 perl.com 网站主编，并为 *The Perl Joural* 撰写专栏，2005 年他所著的 *Higher-Order Perl:Transforming Programs with Programs* 出版。另外，他还是多个 Perl 模块的作者，包括 Text::Template、Memoize、Tie::File 等。——译者注

www.TopSage.com

否则如果该排除的东西没有排除，就有安全隐患了。如果觉得这么做太过谨小慎微，那也可以由我们自己决定限制的严格程度。但无论如何，和外部程序打交道时更苛刻一点是不会错的。

可能你觉的现在已经全部搞定，运行该程序时却发现又碰到了另一个错误：

```
Insecure $ENV{PATH} while running with -T switch
```

由于 Perl 需要去定位 grep 程序所在路径，所以它会从环境变量 $ENV{PATH} 中寻找，但这个环境变量同样也是来自程序外部的，所以它也被认为是受污染的。当 Perl 试图使用该变量时，污染检查就起作用而让程序退出了。为了解决这个问题，我们可以人为将环境变量 $ENV{PATH} 设为空字符串，显然这么一来不存在外部攻入的可能，然后调用外部命令时直接使用它的绝对路径（这样就可以确保调用的就是我们想要的命令）：

```
$ENV{PATH} = '';
...;

print `/usr/bin/grep '$untainted_data' *`;
```

我们也可以直接设置变量$ENV{PATH}为命令所在的绝对路径（由于是我们自己设置的，所以不会被当作污染数据），但请确保这些目录对能运行该程序的用户或组是不可写的[①]，所以这种方式还需要额外处理这方面的工作。

如果想获取关于污染检查更详细的内容，可以阅读 **perlsec** 文档或 *Mastering Perl*[②]的“Security”这章。如果我们想给已有的程序添加污染检查的话，可以先用污染警告模式试试看（见条款 105）。

要点

- 用污染检查可以避免外部数据对程序的影响。
- 用带捕获变量的正则表达式净化数据。
- 在程序中，人为调整环境变量 PATH 的值为可信任的目录。

条款 105　对老旧程序启用污染警告

污染检查是 Perl 开发的好工具，它可以帮助我们找到程序中哪里存在可疑数据传递给外部环境的问题（见条款 104）。一旦启用了污染检查模式，程序外部的所有输入，包括命令行参数、CGI 输入、从文件获取的数据，甚至保存环境变量值的 %ENV 散列等，都会被认为是“受污染”的数据。Perl 将对这类数据作特殊标记。此外，所有由污染数据派生出来的数据也同样被认为是受污染的，所以试图将此类数据传递给外界时，Perl 同样会杀掉我们的程序进程。

对有些年头的老旧程序来说，打开污染检查模式往往就意味着很长一段时间内程序都可能没法正常工作，直到我们找到代码中所有和该模式冲突的地方，并逐一修正、重写或实施其他能解决问题的办法。

如果我们仍希望那些历史遗留的代码成为非污染且安全的代码，可以使用 Perl 的 -t 开关。

① 否则若有人恶意替换掉该目录下的命令，用恶意工具伪装的话，再执行该程序同样会产生安全隐患。——译者注

② brian d foy 所著的 *Mastering Perl*(Sebastopol，CA:O'Reilly Media，2007)。

www.TopSage.com

和它的大哥 -T 开关功能一样，不同的是当它发现问题时只发出警告，不会使程序退出。我们可以在命令行上这样启用：

```
% perl -t program.pl
```

或者，和其他命令行开关一样，在代码的 shebang 行启用：

```
#!/usr/bin/perl -t
```

同样的程序我们换用污染警告来试试：

```
#!perl -t

system("ls -l");
```

污染检查会产生两条警告信息，而不是一出问题就退出程序，此时，程序仍然继续执行：

```
Insecure $ENV{PATH} while running with -t switch...
Insecure dependency in system while running with -t ↵
  switch...
...
```

对存在污染的不安全的老旧代码启用该模式后，我们可以从输出日志中快速定位有问题的地方，然后逐一解决。

在刚才的例子中，污染检查会质疑环境变量 $ENV{PATH} 的设置。因为运行该程序的人可以随意设置他的 PATH 环境变量。因此，程序不能确定它运行的到底是哪个 ls 程序。我们需要将目录的绝对路径设置给 $ENV{PATH} 变量，且要保证运行该程序的用户或组对这些目录没有写入权限。最简单的方法是完全不依赖环境变量$ENV{PATH}，直接指定命令的绝对路径：

```
#!perl -t

$ENV{PATH} = ''; #或者设为'/bin'等其他安全目录

ystem("/bin/ls");
```

现在，两个污染警告就被我们消灭了。

当确信程序已经安全，我们便可以将 -t 开关升级为 -T。虽然已经修复了大多数问题，但某些罕见情况下程序也还是会出问题[①]。

记住，-t 只是用来使遗留代码变得更规范的工具，对新项目请不要使用 -t 开关，而应该直接使用-T 开关。

要点

- 对有些年头的老旧代码启用污染检查要小心。
- 用 -t 开关来给历史遗留的代码启用污染警告模式。
- 对新的代码，应该用 -T 开关启用完整的污染检查。

① 这得看运行时走哪条分支。——译者注

www.TopSage.com

第12章 数据库 12

Perl的DBI模块极大简化了Perl同数据库间的交互，基本上所有流行的（以及许多并不怎么流行的）数据库都能用它来访问。尽管该模块的工作机制十分简单易懂，但使用时仍有许多潜在的高级陷阱需要注意。

数据库交互是一个比较专业的话题，因为它还需要许多Perl以外的相关知识。我们必须了解数据库查询语言、各种数据库服务器专有的特性等超出Perl范畴的事情。有关这个主题铺展开来足以专门写一本书。这里仅简单介绍一些有助于快速掌握Perl操作数据库的基本知识。

我们会故意跳过一些有趣的话题，比如KiokuDB，以及文档数据库比如**CouchDB**，对象关系映射层（ORM，Object Relational Mappers）比如DBIx::Class模块等。这些内容都可以自成一书，如果将来有时间，应该仔细学习研究一下。

条款106 预备SQL语句以复用并节省时间

数据库的查询有许多步骤，先要构造具体的查询语句，接着数据库服务器负责解析该语句，然后按其要求执行查询操作，接着返回我们所需要的结果。这其中任何一步都有可能成为性能瓶颈。

Perl的DBI模块为我们提供了一步到位的做法：

```
use DBI;

my $dbh = DBI->connect(...);

my $rv = $dbh->do('DROP TABLE table');

my $array =
  $dbh->selectall_arrayref('SELECT * FROM table');
```

将类似的selectall_arrayref操作放到循环内部，然后依次取出特定记录的代码可能会是这个样子：

```
foreach my $id (@ids) {
  my $array = $dbh->selectall_arrayref(
    "SELECT * FROM table WHERE id = $id");
}
```


www.TopSage.com

这里给出的查询语句非常简单，只是根据记录编号搜索而已。但想想现实应用中会碰到的情况，一般都要比它复杂得多。联立几张表，用 join 做一些交叉连接，约束一些查询条件，甚至在初步结果上继续进行子查询，复杂度会有所增加。而每次迭代所做的查询，结构上并无变化，只不过约束条件取值不同罢了，但因每次都要从头解析查询语句的语法，并要做好预处理准备，这样累计起来的时耗会很可观。

所以，如果需要重复运行相同的查询语句，哪怕约束条件取值不一样，但只要查询语句结构固定，就可以先缓存对查询语句的解析和预处理结果，以便后续执行时复用。解析和预处理的工作，对于复杂的查询来说可能要费上不少时间。我曾有位同事，写了条无比复杂的查询语句，查询一次要花五分钟才出来结果，虽然已经尽其所能作了优化，但五分钟实在还是有点过长。更糟糕的是，程序中有好几个地方需要运行这条语句，结果更是雪上加霜。最后他终于发现，90%以上的时间都花在了对查询语句的语法语义检查上，但此之后，真正的查询操作却费不了多少时间，甚至可以说相当快。所以他碰到的，根本就不是查询语句的优化问题。

原来一步走，现在拆开来分两步。先预备好查询语句，让它构建和准备好查询，但并不运行。此后，在预备好的基础上运行多少次查询都没关系，省去了每次重复劳动。prepare 方法负责处理所有底层细节，也允许使用占位符（见条款 107）来代表具体查询时将代入数据的位置。prepare 返回的是查询语句句柄对象 $sth，相关的预备信息都存储在其内部：

```
use DBI;
my $dbh = DBI->connect(...)
  or die 'unable to connect to database';

my $sth =
  $dbh->prepare('SELECT * FROM table WHERE id = ?');
```

碰到使用占位符构造的查询语句，数据库会自动缓存。一般优化过的系统都会将解析后的结果，作为执行计划缓存起来。这样后续查询时，只需利用缓存好的执行计划跑一遍就可以了。

这些都准备好之后，在查询语句句柄对象上调用 execute 方法，给出占位符位置上对应的参数值，便能得到查询结果。在循环结构内每次执行查询语句，数据库都会复用缓存的查询语句，免去重复解析，从而保证运行快速：

```
foreach my $id (@ids) {
  my $array = $dbh->execute($id);
}
```

1. 预备多条查询语句应对不同情况

预备好的查询语句，用起来只能完全按照原先计划好的程序走。任何参数数量上的变化，或者查询表的不同，以及别的什么和原来有不一致，都必须重新预备查询语句。当然，预先构造多少查询语句都没问题，待到用时选对句柄就可以了。

比如我们想要取出指定列表中的记录，可以利用 SQL 语句中的 IN 语法：

```
SELECT * FROM table WHERE id IN ( ... )
```

那么，IN 给出的列表中该有多少元素呢？在对查询语句作预备时，只能提供固定数量的占位符，比如下面，必须正好是两个元素：

www.TopSage.com

```
my $sth = $dbh->prepare(
    'SELECT * FROM table WHERE id IN ( ?, ? )'
    );
```

假设我们需要处理 1 到 10 个元素长度的列表查询，由于 DBI 无法自动处理变长参数，我们必须自己逐个构造不同长度的查询语句。下面的代码使用依赖注入的方式（见条款 89）对数据库句柄进行封装。在构造 MyDBI::VariadicPrepare 类的对象实例时，它会创建若干预备查询语句句柄，每一个都对应于参数数量不同的查询，并且我们将各个查询语句句柄都保存在对象散列中：

```
package MyDBI::VariadicPrepare::Cache;

sub new {
  my ( $class, $dbh ) = @_;
  my $self = bless { dbh => $dbh }, $class;
  for my $num ( 1 .. 10 ) {
    $self->{ 'sth_' . $num } = $self->{dbh}->prepare(
      sprintf(
        'SELECT value FROM my_table WHERE id IN (%s)',
        join( ',', ('?') x $num ) )
      );
    }
  return $self;
}
```

相对于直接调用 execute，我们提供了一个封装后的对象方法，只需提供列表元素，多少个都没关系，它会根据具体长度调出合适的句柄，并执行 execute 方法返回结果：

```
sub do_something {
  my ( $self, @values ) = @_;
  if ( @values > 10 ) {
    die 'I can only support 10 values or less';
  }
  $self->{ 'sth_' . @values }->execute(@values);
}
```

这种做法虽然可行，但终归有些粗陋。而且这个例子还限制在不超过 10 个元素长度的查询。再仔细考虑下，情况可能更糟，预先准备好那么多句柄，未必个个都会派上用场。这样的话，所做的无用功仍然是一笔不小的开销，特别是查询语句复杂多变的话。

不必担心，还有更好的办法。

2. DBI 可以帮我们缓存句柄

其实不必自己缓存句柄，DBI 模块本身就提供了 prepare_cached 方法。它会在用户端缓存查询句柄，用起来干净利落，用之前不必费时费力做预备，后续也不用自己维护缓存：

```
use DBI;

my $dbh = DBI->connect(...)
  or die 'unable to connect to database';
my $sth = $dbh->prepare_cached(
  sprintf( 'SELECT value FROM my_table WHERE id IN (%s)',
```

www.TopSage.com

大家网 TopSage.com

```
    join( ',', ('?') x $num ) )
);
```

每次我们调用 prepare_cached 方法，DBI 都会搜索是否已有句柄缓存。如果之前相同的查询语句已经完成了预备工作，就直接返回缓存的版本；否则，就像原来那样新建一个。第一次创建查询句柄时，必须得花些时间等 DBI 和数据库完成相应的初始化预备工作，但这也仅限于实际用到的查询语句，而不用像前例那样对用不到的查询语句也做无用功。

说到这里，必须提醒你注意，如果先后使用同一条查询语句但按不同的参数进行查询，任何一个先调用 finish 方法结束，都会导致另一个查询无法正常工作。这是因为第二个查询所使用的句柄实际就是前一个的缓存，是同一个对象实例，既然它已经关闭结束了，那后续便无法继续取出结果。所以碰到这种情况，就不能用 prepare_cached，应该改成 prepare 方法返回两个完全独立不同的句柄对象，分别进行操作。

3. 并非所有查询语句都需要缓存

有些情况预备 SQL 语句并不会带来多少好处。一般静态的 SQL 查询语句，也就是运行时无需给定参数，不用占位符的语句，在每次直接运行时，数据库会自动使用上次查询后缓存的结果，所以根本不需要作任何预备工作和运行计划。下面两个循环在性能上相差无几：

```
my $query = 'select count(*) from my_table';

foreach ( 1 .. 100 ) {
  my $array = $dbh->selectrow_array($query);
}

foreach ( 1 .. 100 ) {
  my $sth   = $dbh->prepare_cached($query);
  my $array = $sth->fetchrow_array;
}
```

而正因为实际性能差别不大，所以我们也不用担心由于要考虑不必要预备缓存而带来的奇怪情况。

4. 要点

- 通过缓存查询句柄，节约预备工作的重复开销。
- 用 prepare_cached 方法自动缓存用过的查询语句。
- 不必缓存所有查询语句，但即便缓存了也无伤大雅。

条款107　利用 SQL 占位符将参数值自动引起

我们应该养成习惯，始终在 SQL 语句检索值的位置使用占位符。这么做不但能改善性能，还可以提升代码的安全性，使其免于遭受 SQL 注入型的攻击，更何况写起来并不复杂，没理由不用。

在动态构建 SQL 语句时，表达式动态化主要有两个因素：一是查询语句的结构，二是限定

条件的取值。

1. 不要在查询语句中直接内插变量

由于时间紧迫，简单地用 selectall_arrayref 从数据库取数据出来的程序，也许会写成这样：

```
use DBI;

my $dbh->connect( ... );

my $rows = $dbh->selectall_arrayref(
  "select v1, v2 from my_table where id = '$id'"
);
```

这种方式有两处缺陷，除非用于短小程序，否则一定要避免使用。

首先，每次运行 selectall_arrayref，数据库都会做一次完整的新查询。因为多数数据库系统都会严格按查询语句的字面形式，作为提取之前运行计划缓存的依据。如果查询参数的取值一直在变化，显然几乎不会用到缓存。这会让代码的性能显著下降，特别是当查询语句原本就比较复杂，每解析一次都要花几秒种的情况下（见条款 106）。

其次，这种写法容易遭受 SQL **注入**攻击，因为我们无法保证变量 $id 的内容，所以内插替换后形成的新的 SQL 语句，很有可能和原来期望的不同。试想，要是 $id 的值是“1' or 'a'='a”会发生什么情况？代入原来的查询语句我们会发现，这相当于短路了原来的查询，变成始终为真的条件，所以结果会返回 my_table 表中所有的数据。要是 my_table 存着敏感信息，可就麻烦大了。

2. 用占位符表示动态数据的值

要在同一条语句中使用不同的数据分别查询，可以利用**占位符**构造查询语句模板。我们只需在相应位置上填入?，表示稍后执行时再提供具体数据实施查询。

在 selectall_arrayref 中的话，分别给定使用占位符的查询语句、DBI 设置（本例中是空的散列引用）以及本次查询所用数据：

```
my $rows = $dbh->selectall_arrayref(
  'select v1, v2 from my_table where id = ?',
  {}, $id );
```

虽然不同数据库驱动层的工作方式可能有所不同，有些是将所有数据直接发送到服务器处置，有些是按合理的方式转义后代入查询语句，但不管怎样，借助占位符确实能有效避免 SQL 注入的发生。

另外，要是单条语句需要多次运行，也可以将它先做预备，然后在执行时提供具体数据：

```
my $sth = $dbh->prepare(
  'select v1, v2 from my_table where id = ?');

foreach my $id (@ids) {
  $sth->execute($id);
  ...;
}
```


www.TopSage.com

我们甚至可以提前就预备并缓存好相关的查询语句（见条款106）。

3. 创建动态的SQL元素

占位符只适用于代表插入查询语句中的数据值，但它无法表示查询语句的某个组成部分，比如数据库表的名字。若是在运行时才能确定所查询的表，倒是可以利用DBI的quote_identifier方法将表名变量的值安全转义后构造查询语句：

```
my $table = get_table_name();
my $sth   = $dbh->prepare(
  sprintf( 'select * from %s',
    $dbh->quote_identifier($table) )
);
```

但它不会检查是否确实存在$table表。如果我们使用的数据库驱动支持table_info方法，可以借此判定表是否存在。更多详细信息请参考DBI的文档。

4. 要点

- 不要在SQL语句中内插变量。
- 查询数据库时，用占位符来接受实际的数据。
- 占位符仅支持数据替换，无法用作查询语句中的标识符。

条款108　通过绑定返回列快速访问数据

从DBI句柄提取数据时，一般我们会把对应值赋给变量。这种做法浅显直白，一看就知道哪个变量包含的是哪个字段的值：

```
use DBI;

my $dbh = DBI->connect(...);

my $sth =
  $dbh->prepare('select one, two, three from my_table');
$sth->execute;

while ( my ( $one, $two, $three ) = $sth->fetchrow_array )
{
    ...;
}
```

但这样做有些低效，只是处理几条记录的话并不成问题，可如果遇到成千上万条记录，或许你就能体会到差别了。赋值操作虽然简单，但累计起来总归是一笔不小的开销。

其实我们直接将固定变量的引用**绑定**到查询句柄上，通过类似内存指针的方式定位数据，就可以避免无效的赋值操作。实际上，这是让数据库往预先设定好的变量写数据而已，每次循环都将每行中的每列数据依次填充到对应位置。我们可以用bind_columns方法来选择我们需要的列：

```
use DBI;
```

www.TopSage.com

```
my $dbh = DBI->connect(...);

my $sth =
  $dbh->prepare('select one, two, three from my_table');
$sth->execute;

my ( $one, $two, $three );
$sth->bind_columns( \$one, \$two, \$three );
while ( $sth->fetchrow_arrayref ) {
  ...;
}
```

一下传入一个变量引用列表的写法貌似多有不便，可读性也有所欠缺。我们可以换种写法一次绑定一个。bind_col 方法接受两个参数，一个是从 1 开始计算的字段位置，一个是要绑定的变量引用。每次 execute 完成后都应该调用 bind_col：

```
use DBI;

my $dbh = DBI->connect(...);

my $sth =
  $dbh->prepare('select one, two, three from my_table');
$sth->execute;

$sth->bind_col( 1, \my $one );
$sth->bind_col( 2, \my $two );
$sth->bind_col( 3, \my $three );

while ( $sth->fetchrow_arrayref ) {
  ...;
}
```

除此之外，我们还可以更好地控制 bind_col 的行为。如果知道特定列的数据类型，可以在绑定时提供第三个参数，作为相关属性列出。用 DBI 模块导出标签为 :sql_types 的一系列常量到当前命名空间，便可以在代码中直接选用这些数据类型：

```
use DBI qw(:sql_types);

$sth->bind_col( 1, \my $one, { TYPE => SQL_DATETIME } );
```

作为捷径，如果只有一个属性，第三个参数可以直接写成标量：

```
use DBI qw(:sql_types);

$sth->bind_col( 1, \my $one, SQL_DATETIME );
```

绑定之后，数据的访问必然会快上许多，你应该能感觉得到。

要点

- 赋值到变量是一种略显低效的做法。
- 按列绑定返回数据到变量引用，可以更快地访问返回的数据。
- 如果知道数据类型，可以在绑定时指定。

www.TopSage.com

条款109　复用数据库连接

连接数据库是一项比较复杂耗时的操作，我们应该避免创建过多的连接，尽可能使用缓存的连接。这样不但能减少查询所需花费的时间，还能为其他用户保留可用连接。对于如今随处可见的高并发大流量网站来讲，这点尤其重要。我们不光要注意避免同一个程序内部的多次连接，也应当避免使这样的程序每分钟跑上几百次。

1. 连接数太多

好多程序员都没有将他们代码中数据库特性这部分纳入更高级设计的范畴。或许是因为程序开始时还很小，无需苛责，但随着程序功能的演进，仍然对刚开始时编写的粗陋代码听之任之，就说不过去了。比如下面这段代码，仅是为了一次查询，特地做了一次专门的数据库连接，用完后就随手关闭抛弃了：

```
sub find_value {
  my ($id) = @_;
  my $dbh =
    DBI->connect( $dsn, $user, $password, %options );
  my ($value) = $dbh->selectall_array(
    'select value from my_table where id = ?',
    {}, $id );
  return $value;
}
```

一两次调用 find_value 的话，还没什么影响，但次数一多，程序就会花上数倍时间用在连接数据库上面，而这种无谓开销是完全可以避免的。

2. 共享数据库连接

与其每次查询都特地建立数据库连接，不如连接一次然后缓存数据库句柄。具体做法得看应用程序的实际需求。不过一般我们会把连接句柄作为对象数据保存起来：

```
package MyApp::Foo;

sub new {
  my ($class) = @_;
  my $self = bless {}, $class;
  $self->{dbh} = DBI->connect(...);
  return $self;
}
```

然后在需要查询数据库时，从对象获取该连接句柄：

```
sub find_value {
  my $self = shift;
  my ($value) =
    $self->{dbh}->selectall_array(
    'select value from my_table where id = ?',
    {}, $id );
  return $value;
}
```

www.TopSage.com

这个办法对运行时间很短的普通程序而言还是不错的，但对长时间运行的程序就会有问题。如果一段时间没有交互，数据库服务器会切断连接，抑或网络通信出现故障拥堵，等等，种种不确定因素都有可能导致原先缓存的连接句柄失效。而我们在写程序时，却始终假想数据库已经开启并可以使用。

● 一个对象一个连接

解决办法是将数据库连接句柄封装起来，改由特定对象方法返回句柄，而不再是直接取用对象数据。作为用户，你不必关心该方法到底做些什么以及怎么做，只需知道它能给你一个可用的数据库连接句柄即可。请看下面的代码，对象方法 get_dbh 借用 DBI 的 ping 方法检查当前连接是否有效，如果 ping 返回假，则重新连接并缓存新的句柄：

```
package MyApp::Foo;

sub connect_to_database {
  DBI->connect(...) or die ...;
}

sub get_dbh {
  my ($self) = @_;

  unless ( $self->{dbh}->ping ) {
    $self->{dbh} = $self->connect_to_database;
  }

  $self->{dbh};
}
```

因为每个对象各自要维护一个数据库连接，所以这个方法多少还有些资源浪费，不够完善。

● 一个类一个连接

如果让每个对象实例都共享同一个数据库连接，可以把句柄存为类数据。其基本实现同上，只不过把连接句柄存到类级别的词法变量：

```
package MyApp::Foo;

my $dbh;

sub connect_to_database {
  DBI->connect(...) or die ...;
}

sub get_dbh {
  my ($self) = @_;

  unless ( $dbh->ping ) {
    $dbh = $self->connect_to_database;
  }

  $dbh;
}
```

请注意，$dbh 是一个源文件范围内的全局变量，所有该文件中的代码都可以访问它，哪怕

切换到其他包也是如此。

● 再进一步

有时候，就算一个类一个连接也还是会出现资源浪费，那就改成整个应用程序共享同一个连接，或者在多个进程间共享。超大型的应用或许还得用上一套专门的数据库连接管理系统。

Apache::DBI 是一个比较典型的在不同进程间共享数据库连接的模块。它在幕后封装接管 DBI 的内部实现，并返回缓存后的数据库连接。就算不用 Apache 也可以借用它的代码，从 Apache::DBI 支持 Apache2 之前的版本当中提取，移植为符合我们自己需要的样子，应该不会很难。

除此之外，还可以考虑用 DBD::Gofer 模块架设代理服务器，由它管理 DBI 连接。它会在独立进程中维护连接缓存，不但灵活，还具伸缩性，也能管理流量等诸多事情。设置妥当后，只需在代码中修改 DBI->connect 部分，使用 DBD::Gofer 代理服务器即可：

```
use DBI;

my $normal_dsn = "dbi:...";

my $dbh = DBI->connect(
  "dbi:Gofer:transport=$transport;...;dsn=$original_dsn",
  $user, $passwd, \%attributes
);
```

这里要小心新的数据源名称：前半部分是为 DBD::Gofer 而设，抽取掉这部分后，DBD::Gofer 会利用 dsn 指定的内容建立实际的数据库连接，而这个“普通的”非 Gofer 数据源 dsn 需要写在最后面。

从 DBD::Gofer 拿到数据库连接句柄后，使用方式便和以往没什么两样。程序的其他部分却根本不会察觉到有什么区别。

3. 要点

- 不要每次查询时都连接数据库。
- 在对象、类的内部或进程间共享数据库连接。
- 用 DBD::Gofer 模块作为 DBI 代理，共享连接。

www.TopSage.com

第13章 杂 项 13

林林总总的还有许多内容无法安排在之前的章节中，但这些主题又非常值得详加阐释，所以并在这里作为杂项列出。其中有些主题还有延展拓宽的余地，将来或许能专门成章，另外一些相对比较独立的主题，也同样非常有用。总之，我们觉得这些主题都很重要，应该有所了解。

条款110 编译并安装自己的perl解释器

Perl的流行和成功也带来了一些麻烦。基本上每个操作系统，不管是UNIX还是Linux体系的，甚至包括Mac OS X，各自都有一份内置的perl解释器。许多系统把Perl当作日常事务处理不可或缺的一部分，就算默认没有安装一般也会提供perl的安装包。不管怎样，几乎所有的平台都有预先编译好的perl可以直接拿来使用。

一般在Windows系统上，大家都会选用预编译好的二进制版本，比如ActiveState（http://www.activestate.com/），或者cygwin（http://www.cygwin.com/），或者Strawberry Perl（http://www.strawberryperl.com/）。有了这些选择，人们再也不必自己动手编译了。

但话说回来，自己编译一份perl还是有很多好处。由于许多操作系统的日常维护会依赖于当前的perl程序，所以最好不要弄乱它。如果要升级的某个核心模块恰好用于操作系统的关键业务，万一不兼容或者有缺陷，就会导致系统故障。尽量无视系统内建的perl好了，不去改动它总归没错的。

另外，作为一名Perl程序开发者，确实需要安装一些不同版本的perl，以便在不同环境下测试。安装和维护各个独立版本并不困难，接下来我们就会讨论。但涉及与你的实际系统相关的特殊需求，还请参考README文件中的相关指引。

1. 编译自己的perl

编译perl需要C编译器及其他相依赖的构建工具。此外还会用到make命令，或其类似的变种版本。编译时无需特殊权限，最终安装时可以放到自己的用户主目录中。本条款假设我们已经准备好这些工具，随时可以动手编译。

先下载一份所需perl版本的源代码包，具体地址可以到CPAN上找（http://www.cpan.org/src/README.html）。

www.TopSage.com

解开压缩包后，进入该目录。现在我们得决定安装位置。这里作为例子，我们把各个版本的perl统一放到/usr/local/perls目录下，这样每个子目录下就是一份独立的perl版本。

接下来要用到的Configure脚本会检查系统环境，并为编译perl源代码做好准备。-des开关项表示对所有配置选项的问题都按默认选择。-D开关作为补充，用来指定默认以外的配置，在下面的例子中，我们另外设定了表示安装位置的prefix参数。

```
% ./Configure -des -Dprefix=/usr/local/perls/perl-5.10.1
```

这样的话，就设置为把perl安装到/usr/local/perls/perl-5.10.1目录下。因为默认不会将此版本的perl链接到/usr/bin中，所以不必担心弄乱现有系统。另外建议你还是用普通用户身份做这些事，万一碰到伤筋动骨的操作，也会因为权限限制而有所规避。

如果好奇，想看看都有哪些选项可以配置，去掉默认接受的开关项，跟着提示从头到尾走一遍配置流程（一生当中总要试一次吧）：

```
% ./Configure
```

跑完Configure后，不管之前配了些什么，都可以开始编译源代码了。按操作系统不同，我们需要一个make命令（或其变种）：

```
% make all
```

编译过程结束后还得测试，这个步骤比较费时：

```
% make test
```

完成之后便可以安装了。你会看到它将编译得到的perl及相关文件复制到之前prefix指定的目录中：

```
% make install
```

安装完毕后，可以再试试安装另一版本，重新配置，比如启用多线程特性，然后指定安装到特定目录：

```
% ./Configure -des -Dusethreads ↵
  -Dprefix=/usr/local/perls/perl-5.10.1-threaded
```

2. 使用perl

特定版本的perl安装好之后就算告一段落了，不用再做什么配置，最多是使用的时候得给出完整的perl程序路径罢了。要查看默认的模块搜索路径，可以输入：

```
% /usr/local/perls/perl-5.10.1/bin/perl -V
```

所有相关的工具、外部程序以及内建的模块源文件等，都会被安置到这个新安装的perl所在的目录中。如果要用cpan工具为此版本的perl安装模块，就必须用它自己的cpan程序。该程序位于指定路径下的bin/目录中：

```
% /usr/local/perls/perl-5.10.1/bin/cpan LWP::Simple
```

任何以这种方式安装的模块，都将被安置于该版本perl所在目录下，而不会影响其他版本。所以，这是多版本perl共存的关键。自然地，切换到其他版本的perl后，所需模块得再安装

www.TopSage.com

一次。

阅读文档也是一样，得指定正确的 perldoc 的路径，否则无法定位正确的模块路径：

```
% /usr/local/perls/perl-5.10.1/bin/perldoc ↵
  LWP::Simple
```

如果想把该版本的 perl 作为默认使用的版本，通过添加它的路径到环境变量 PATH 便可以省去麻烦的完整路径写法。这种方式依旧保留了系统提供的 perl，但同时也便于我们使用自己的选择。什么时候要切换回去，只需更新环境变量就行，shell 会依此使用路径中最先找到的那个 perl 版本，包括其下的模块搜索路径。

不过，在用 CPAN.pm 或 CPANPLUS 时，可能会有点小麻烦。因为它们保存配置信息时，是放在用户主目录中的（见条款 65）。所以这些配置信息是按用户，而不是按 perl 版本区分的。所以如果要用这些工具的话，务必在使用之前先重新配置一遍，以免出问题。

3. 要点

- 安装自己的 perl 版本，以保证系统 perl 的独立和干净。
- 不同的配置选项，不同的版本，都可以安装到各自目录下，实现多版本 perl 共存。
- 常用的 perl 版本，可以将其路径添加到 PATH 环境变量中，作为默认版本使用。

条款 111　用 Perl::Tidy 美化代码

易于维护的代码，必然是要用统一且一致的编码风格。有规律的缩进和对齐，会让代码看起来更清晰明了。当然，我们不用亲自手工排布美化代码格式，借助 Perl::Tidy 便可实现。

先准备一份格式杂乱的代码，存为 ugly.pl：

```
use warnings; use strict;
while(<>) { if(/\d/
) { print "contains number\n"; }
else { print "number-free\n" } }
```

运行 perltidy 处理该文件：

```
% perltidy ugly.pl
```

perltidy 命令会根据原来的文件名，将重新格式化后的代码存入后缀为 .tdy 的文件。以下是 ugly.pl.tdy 中的结果：

```
use warnings;
use strict;
while (<>) {
  if (/\d/) { print "contains number\n"; }
  else      { print "number-free\n" }
}
```

perltidy 不会覆盖掉原来的 ugly.pl 文件，以保证不破坏原来的程序。尽管 perltidy 只是重新规排格式，应该不会引入什么错误，但也不能就此掉以轻心，格式化后再跑一次编译检查，甚至是所有的单元测试，确认没有问题之后，再删除原来的文件也不迟（当然，最好还是用版本

www.TopSage.com

大家网 TopSage.com

控制系统管理源代码，会比较稳妥）。

如果图方便，可以用 -b 选项，让它创建一个.bak 后缀的文件保存原始的文件内容，然后把经过格式化的代码替换掉当前文件。这样，后续的测试仍旧可以按原来的方式进行，不用再重新组织文件，但备份的原始代码还得根据需要另行保存。

1. Perl::Tidy 的配置

如果我们想改变格式化的样式，它的命令行方式提供了许多格式化细节选项，比如把默认四个空格的缩进改为两个，把结果打印到标准输出而非默认的 .tdy 文件，可以这样使用：

```
% perltidy -st -i=4  ugly.pl
```

现在我们的 ugly.pl 看起来有稍许不同（会被显示在终端屏幕上）：

```
use warnings;
use strict;
while (<>) {
    if (/\d/) { print "contains number\n"; }
    else      { print "number-free\n" }
}
```

如果不喜欢起始的花括号跟在行尾，试试这个：

```
% perltidy -st -bl -bli ugly.pl
```

现在花括号和 while 各占一行：

```
use warnings;
use strict;
while (<>)
  {
    if (/\d/) { print "contains number\n"; }
    else      { print "number-free\n" }
  }
```

确定了我们喜欢的格式之后，不用每次都在命令行打一遍，把它们存到配置文件 .perltidyrc 中，即可一劳永逸。做法很简单，另外给出 -dump-options 选项就能完成：

```
% perltidy -st -bl -bli -dump-options
```

现在，不光是当前给定的格式化选项，还包括其他默认设置，都会一并存入该配置文件：

```
# Final parameter set for this run.
# See utility 'perltidyrc_dump.pl' for nicer formatting.
--add-newlines
--add-semicolons
--add-whitespace
--backup-file-extension="bak"
--blanks-before-blocks
--blanks-before-comments
--blanks-before-subs
...
```

此外，还能为特定的项目创建一个专有的配置档，使用时指定 --profile 即可：

```
% perltidy --profile=projecttidy ugly.pl
```

不仅如此，我们还能定义一个名为 PERLTIDY 的环境变量，藉由该变量指定的配置文件，覆盖默认定义在 /usr/local/etc/perltidyrc 或 /usr/local/etc/perltidyrc 中的配置。

配置好之后，如果要临时取消应用该配置，只需要在命令行提供 --noprofile 开关即可：

```
% perltidy --noprofile ugly.pl
```

2. 语法检查

就算代码存在语法问题，Perl::Tidy 也是可以正常工作的，但不总是如此，碰到实在令其困惑的情况，它会另行创建一个 .ERR 的文件列出错误所在。

下面是一则明显存在语法问题的代码，因单引号的使用不当导致出现了歧义：

```
# b0rk3n.pl
use warnings;
use strict;

print 'I'm broken';
```

于是，在运行 perltidy 时，它会报告有问题，并把这些问题存入 b0rk3n.pl.ERR 文件：

```
% perltidy b0rk3n.pl
## Please see file b0rk3n.pl.ERR
```

以下就是错误文件中的内容：

```
5: print 'I'm broken';
         ---^
found m where operator expected (previous token ↵
    underlined)
5:       hit EOF seeking end of quote/pattern starting ↵
    at line 4 ending in b
```

虽然实际上错在单引号没有正确转义，但 perltidy 却认为字符串定义结束，后续的 m 是匹配操作的开始，但接着又无法解释后续的语法。所以我们不必死抠它的报错消息，看看出错处的上下文，多半能找出实际的问题。

3. 在测试环节自动格式化

我们可以在每次运行作者自己的测试集合（见条款 88）时，自动运行 perltidy 格式化代码。此项工作可交由 Test::PerlTidy 模块负责：

```
use Test::More;

plan skip_all =>
  'Set $ENV{TEST_AUTHOR} to enable this test.'
  unless $ENV{TEST_AUTHOR};

eval "use Test::PerlTidy";
plan skip_all => 'Test::PerlTidy required' if $@;

run_tests();
```

自然，默认情况下 Test::PerlTidy 会使用标准选项。如果要自行配置，就得在模块根目录或自己的用户主目录中配置好 .perltidyrc 文件。

www.TopSage.com

4. 要点

- 代码风格要统一，这样才有助于后续阅读和维护。
- 用 Perl::Tidy 模块自动格式化代码。
- 在.perltidyrc 文件中配置自己喜好的风格。

条款112 使用 Perl Critic

Perl 以其高度的灵活性和表现形式的多样性著称于世（又或者说臭名远扬）。长久以来，在 Perl 程序员之间流传最广的一句信条就是："条条大路通罗马。"（There's More Than One Way To Do It，通常简写为 TMTOWTDI。）实现方式灵活固然好，但有时为了稳妥还是得循规蹈矩，照着编码规范行事。特别是在团队分工协作时，要决断某些随性写法的取舍，多半见仁见智，既要避免写出晦涩难懂的代码，又要保证代码形式在一定程度上的统一。即便项目只有你一个人，最好也还是遵循若干编码规范，以保持代码的连贯和一致，这样做总没什么坏处。

有无规范是一件事，是否强制执行又是另一回事。我们可以采取敏捷开发的方式，比如代码互审、结对编程，这都很好，但却仍无法完全避免人为因素的疏漏。所以我们真正需要的，是一个自动化的系统，由它负责审阅代码并指出什么地方不合要求。

Perl Critic 就是一个这样的系统。它对给定的 Perl 源代码做静态分析，发现问题就输出警告。虽然它本身无法自动解决所有问题，但至少对我们管理代码有所帮助。

1. 在互联网上

最方便的方法是将源文件上传到 http://perlcritic.com/，由在线系统检查是否符合标准的 Perl Critic 规约。什么都不用安装，先试试 Perl Critic，然后再决定是否要用它。网页上的输出基本和命令行方式下的输出一致，具体请看下一节。

2. 在命令行方式下

安装好 Perl::Critic 之后，便可以直接在命令行下使用 perlcritic 这个程序了。

这里有一段简短的 Perl 程序，按照 Perl Critic 的默认配置，它会触发几条警告信息：

```
package MyPrinter;

sub new {
  bless {}, shift;
}
sub print {
  print "Hello, World\n";
}

1;
```

这样一段简单的模块代码，perlcritic 还是找出了其中潜在问题。它警告说，没有启用 strict（见条款3）：

```
perlcritic MyPrinter.pm
```

www.TopSage.com

```
  Code before strictures are enabled at line 3, column 1.↵
    See page 429 of PBP. (Severity: 5)
```

在错误消息中，perlcritic 会指出问题所在行的行号。此外，如果这个问题在 Damian Conway 的 *Perl Best Practices*[①] 一书中涉及的话，它也会一并给出该书中相关内容的页码。最后，perlcritic 会对此问题的 severity（严重程度）评级。

代码问题的严重程度共分五级，5 分最重，1 分最轻。一般看到 5 分的警告就该立即修正。默认 perlcritic 只显示级别为 5 的问题，要查看低级别的问题，可以使用 --severity 开关选项：

```
% perlcritic --severity 1 MyPrinter.pm
Error: No word lists can be found for the language "↵
  en_US".
Code is not tidy at line 1, column 1. See page 33 of ↵
  PBP. (Severity: 1)
RCS keywords $Id$ not found at line 1, column 1. See ↵
  page 441 of PBP. (Severity: 2)
RCS keywords $Revision$, $HeadURL$, $Date$ not found at ↵
  ine 1, column 1. See page 441 of PBP. (Severity: 2)
RCS keywords $Revision$, $Source$, $Date$ not found at ↵
  line 1, column 1. See page 441 of PBP. (Severity: 2)
No "$VERSION" variable found at line 1, column 1. See ↵
  page 404 of PBP. (Severity: 2)
Subroutine "new" does not end with "return" at line 3, ↵
  column 1. See page 197 of PBP. (Severity: 4)
Code before strictures are enabled at line 3, column 1. ↵
  See page 429 of PBP. (Severity: 5)
Code before warnings are enabled at line 3, column 1. ↵
  See page 431 of PBP. (Severity: 4)
Subroutine name is a homonym for builtin function at ↵
  line 7, column 1. See page 177 of PBP. (Severity: 4)
Subroutine "print" does not end with "return" at line 7, ↵
   column 1. See page 197 of PBP. (Severity: 4)
Return value of flagged function ignored - print at line ↵
   8, column 5. See pages 208,278 of PBP. (Severity: 1)
```

Perl::Critic 把一系列规约集合到一起形成可选的主题（theme），我们可以指定主题名字做不同策略的检查。比如说，按照 *Perl Best Practices* 一书中提及的规约检查，可以指定使用 pbp 主题：

```
% perlcritic --brutal --theme=pbp
```

我们也可以自己修订主题，比如大体上沿用 pbp 主题，但跳过有关代码格式修饰或是处理维护发布过程的部分，可以直接在命令行调整：

```
% perlcritic --brutal --theme=\
  'pbp && ! (cosmetic || maintenance)' MyPrinter.pm

No "$VERSION" variable found at line 1, column 1. See ↵
```

① Damian Conway 所著的 *Perl Best Practices*（Sebastopol，CA：O'Reilly Media，2005）。

www.TopSage.com

```
 page 404 of PBP. (Severity: 2)
Subroutine "new" does not end with "return" at line 3, ↵
 column 1. See page 197 of PBP. (Severity: 4)
Code before strictures are enabled at line 3, column 1. ↵
   See page 429 of PBP. (Severity: 5)
Code before warnings are enabled at line 3, column 1. ↵
   See page 431 of PBP. (Severity: 4)
Subroutine name is a homonym for builtin function at ↵
   line 7, column 1. See page 177 of PBP. (Severity: 4)
Subroutine "print" does not end with "return" at line 7, ↵
   column 1. See page 197 of PBP. (Severity: 4)
```

perlcritic 提供了诸多命令行选项，我们可以用正则表达式灵活地约束或包含某些策略主题，指定最终输出的建议数量，或者自定义输出格式，显示代码统计报告等。任何在命令行可用的配置选项都可写入.perlcriticrc 文件，这样就不必重复输入，并且效果却完全一样。花些时间看看有关 perlcritic 的文档，必定获益良多。

3. 在测试组件中

只要能熟记各种选项，玩转 Perl::Critic 的命令行界面自然不在话下。不过，我们也可以借助它的另一个测试模块接口 Test::Perl::Critic，在每次运行测试时自动做静态分析。许多人都习惯把有关该模块的配置选项写在 t/perlcriticrc 文件中随同模块一起发布，然后再添加一个 t/perl-critic.t 模板文件：

```
use File::Spec;
use Test::More;

if ( not $ENV{TEST_AUTHOR} ) {
  plan( skip_all =>
    'Set $ENV{TEST_AUTHOR} to a true value to run.' );
}

eval "use Test::Perl::Critic";

if ( $@ ) {
  plan( skip_all =>
    'Test::Perl::Critic required to criticise code' );
}

my $rcfile = File::Spec->catfile( 't', 'perlcriticrc' );
Test::Perl::Critic->import( -profile => $rcfile );
all_critic_ok();
```

有时在项目开发后期才加入 Test::Perl::Critic，会看到一下子涌出无数问题需要修正。为了不影响后续开发进程，可以改用 Test::Perl::Critic::Progressive 暂时搁置之前有问题的报告，它只针对新出现的问题给出建议：

```
use Test::More;

eval {
  use Test::Perl::Critic::Progressive
```

www.TopSage.com

TopSage.com

```
    qw( progressive_critic_ok )
};
plan skip_all =>
    'T::P::C::Progressive required for this test' if $@;

progressive_critic_ok();
```

每次运行 Test::Perl::Critic::Progressive 时它都会标记当前遇到的所有代码，以便下次运行时跳过分析检查，这样一来，每次只报告自上次成功运行测试以来新增代码所出现的问题。这种逐步递进的方式更容易让人接受，而逐条修正一样可以保证代码质量。

我们甚至可以按不同策略单独设置报告问题的数量，以加快与特定策略相关的问题的修正速度：

```
use Test::More;

eval {
  use Test::Perl::Critic::Progressive
    qw(
      progressive_critic_ok
      set_total_step_size
    )
};
plan skip_all =>
    'T::P::C::Progressive required for this test' if $@;

set_total_step_size( 5 );
progressive_critic_ok();
```

由于 Test::Perl::Critic::Progressive 的默认做法只是检查上次成功运行测试以来出现的问题，所以报告的问题一般不会很多，但为了促使你和你的团队不断减少以前代码中出现的问题，可人为设置增加每次 Test::Perl::Critic::Progressive 报告的问题数目：

```
my %step_sizes = (
    'ValuesAndExpressions::ProhibitLeadingZeros' => 2,
    'Variables::ProhibitConditionalDeclarations' => 1,
    'InputOutput::ProhibitTwoArgOpen'            => 3,
);

set_step_size_per_policy( %step_sizes );
progressive_critic_ok();
```

4. 自定义策略

Perl::Critic 模块本身就提供了许多默认策略，而除此之外，CPAN 上也有许多非核心的策略可以集成到我们的系统中来，以满足某些特定需求。不过这样的第三方策略为数众多，不一而足，这里仅挑选了几个相对值得关注的：

- Perl-Critic-More

 由 Perl::Critic 的开发者编写的策略集合包。

- Perl-Critic-Bangs

 由 Andy Lester 编写的一组策略，可以搜索注释行中的代码和标记以及糟糕的变量命名。

www.TopSage.com

- Perl-Critic-StricterSubs
 设定了若干比核心模块更为严格的、检查子程序用法方面的规则，适用于那些对代码要求高到近乎病态的开发者。
- Perl-Critic-Swift
 要求在源代码及 Pod 文件中声明使用 UTF-8 编码的策略。
- Perl-Critic-Moose
 针对基于 Moose 开发的代码的检查策略。

如果实在找不到所需策略，不妨自己写一个。这并不难，阅读下 Perl::Critic 模块中的 Perl::Critic::DEVELOPER 文档，大体就可以开始动手了。

5. 要点

- 用 Perl::Critic 检查代码中潜在的问题。
- Perl::Critic 一并提供了用于测试的 Test::Perl::Critic 组件。
- 用 Test::Perl::Critic::Progressive 模块对现有项目加入 Perl Critic 检测。

条款113　用 Log::Log4perl 记录程序运行状态

适当的日志信息记录，对后期的代码调试和诊断来讲，可说是一种强大的、不可或缺的工具。最基本的日志记录，可以只是简单地使用 print 语句：

```
print "The value is [$value]\n";
```

稍加改进，通过环境变量控制是否输出日志，可能会更好些：

```
print "The value is [$value]\n" if $ENV{DEBUG};
```

1. 易用的 Log::Log4perl

上面的做法太过随意，也不够灵活。真正要体现日志记录的强大之处，还是得选用专业的日志记录框架，比如 Log::Log4perl，它会负责处理所有底层细节。要初始化 Log::Log4perl，最便捷的方式就是用它提供的 easy_init 方法。引用模块时指定 :easy 标签就会导入表示不同记录级别的若干变量以及相应的子程序：

```
use Log::Log4perl qw(:easy);
Log::Log4perl->easy_init($ERROR);

INFO 'starting program';

for ( 1 .. 100 ) {
  DEBUG "loop counter: $_";
}

INFO 'program complete';
```

但运行这段代码，根本就看不到任何输出！Log::Log4perl 会直接忽略级别低于 easy_init 中指定的那些日志输出。

www.TopSage.com

Log::Log4perl 中定义了以下这些记录级别：

- FATAL
- ERROR
- WARN
- INFO
- DEBUG
- TRACE

每个级别都有对应同名的变量，比如 $ERROR，以及同名的子程序。初始化 Log::Log4perl 之后，它只输出指定级别或更高级别的那些日志。上面的例子中我们初始化为 $ERROR 级别，所以最终只有 ERROR 和 FATAL 级别的日志才会真正输出。如果要获得更多的输出（即扩大输出的范围），可以修改到低一些的级别：

```
Log::Log4perl->easy_init($INFO);
```

现在能看到有输出了，不过只有来自 INFO 级别的：

```
2009/10/20 15:47:45 starting program
2009/10/20 15:47:45 program complete
```

试着改为更低一些的级别：

```
Log::Log4perl->easy_init($DEBUG);
```

现在可以看到多了不少 DEBUG 级别的输出：

```
2009/10/20 15:48:16 starting program
2009/10/20 15:48:16 loop counter: 1
2009/10/20 15:48:16 loop counter: 2
2009/10/20 15:48:16 loop counter: 3
...
2009/10/20 15:48:16 loop counter: 98
2009/10/20 15:48:16 loop counter: 99
2009/10/20 15:48:16 loop counter: 100
2009/10/20 15:48:16 program complete
```

2. 面向对象方式的接口

Log::Log4perl 还提供了面向对象方式的接口，使之用起来更为灵活方便。继续用 easy_init 初始化好了，然后调用类方法 get_logger 返回日志对象。使用时在该对象上调用同名方法即可，只不过名字要全部改为小写：

```
use Log::Log4perl qw(:easy);
Log::Log4perl->easy_init($DEBUG);

my $logger = Log::Log4perl->get_logger();

$logger->info('starting program');

for ( 1 .. 100 ) {
  $logger->debug("loop counter: $_");
}
```


www.TopSage.com

```
$logger->info('program complete');
```

3. 更好的配置

用过一段时间后，我们可能会对 easy_init 提供的默认选项不够满意，想自己做些配置。可以调用类方法 init 来加载自己定义的配置文件：

```
use warnings;
use strict;
use Log::Log4perl;
Log::Log4perl::init('log4perl.conf');

my $logger = Log::Log4perl->get_logger();

$logger->info('starting program');

for ( 1 .. 100 ) {
  $logger->debug("loop counter: $_");
}

$logger->info('program complete');
```

由于 Log::Log4perl 相当于 Log4j 的 Perl 版本，所以它的配置文件看起来和 Java 的配置文件十分类似。我们可以自由选择**目标载体**（appender），指定输出日志的最终流向，也可以自定义输出信息的格式。通过定义 rootLogger 指定最低输出级别以及日志记录的目标载体：

```
log4perl.rootLogger=DEBUG, Screen

log4perl.appender.Screen=Log::Log4perl::Appender::Screen
log4perl.appender.Screen.stderr=1
log4perl.appender.Screen.layout= ↵
  Log::Log4perl::Layout::SimpleLayout
```

再次运行程序，会看到新的输出格式：

```
INFO - starting program
DEBUG - loop counter: 1
DEBUG - loop counter: 2
DEBUG - loop counter: 3
...
DEBUG - loop counter: 98
DEBUG - loop counter: 99
DEBUG - loop counter: 100
INFO - program complete
```

但还有改进的余地，同时定义多个目标载体，便可以将日志同时输送到不同的地方：

```
log4perl.rootLogger=DEBUG, Screen, Logfile

log4perl.appender.Logfile=Log::Log4perl::Appender::File
log4perl.appender.Logfile.filename=my_program.log
log4perl.appender.Logfile.mode=replace

log4perl.appender.Logfile.layout=PatternLayout
```

www.TopSage.com

```
log4perl.appender.Logfile.layout.ConversionPattern= ↵
  [%r] %F %L %c - %m%n

log4perl.appender.Screen=Log::Log4perl::Appender::Screen
log4perl.appender.Screen.stderr=1
log4perl.appender.Screen.layout= ↵
  Log::Log4perl::Layout::SimpleLayout
```

现在，运行时的日志信息会被同时记录到 my_program.log 文件中，且选用的是另一种格式：

```
[33] log4perl_object_with_config.pl 8 main ↵
    -starting program
[33] log4perl_object_with_config.pl 11 main ↵
   - loop counter: 1
[33] log4perl_object_with_config.pl 11 main ↵
    -loop counter: 2
[33] log4perl_object_with_config.pl 11 main ↵
    -loop counter: 3
...
[44] log4perl_object_with_config.pl 11 main ↵
    -loop counter: 98
[45] log4perl_object_with_config.pl 11 main ↵
    -loop counter: 99
[45] log4perl_object_with_config.pl 11 main ↵
    -loop counter: 100
[45] log4perl_object_with_config.pl 14 main ↵
    -program complete
```

我们甚至还可以创建不同的日志记录类型 (category)，以修订现有的目标载体、记录格式等。每个类型都能定义各自的记录级别，也能自由独立地启用或关闭，甚至可以输出到不同的目标载体：

```
package MyFirstClass;

use warnings;
use strict;
use Log::Log4perl;

Log::Log4perl::init('log4perlWithClasses.conf');

sub new {
  my $logger = Log::Log4perl->get_logger(__PACKAGE__);
  $logger->debug( 'creating new ', __PACKAGE__ );
  return bless {}, shift;
}

package MySecondClass;

use warnings;
use strict;
use Log::Log4perl;

Log::Log4perl::init('log4perlWithClasses.conf');
```


www.TopSage.com

大家网 TopSage.com

```
sub new {
  my $logger = Log::Log4perl->get_logger(__PACKAGE__);
  $logger->debug( 'creating new ', __PACKAGE__ );
  return bless {}, shift;
}

package main;

my $first  = MyFirstClass->new();
my $second = MySecondClass->new();
```

在配置文件中，依次设置不同的类型：

```
log4perl.rootLogger=WARN, Screen

log4perl.category.MyFirstClass=ERROR

log4perl.category.MySecondClass=DEBUG

log4perl.appender.Screen=Log::Log4perl::Appender::Screen
log4perl.appender.Screen.stderr=1
log4perl.appender.Screen.layout= ↵
  Log::Log4perl::Layout::SimpleLayout
```

再次运行程序，会看到漂亮的调试输出内容：

```
DEBUG - creating new MySecondClass
```

4. 探测当前记录级别

Log::Log4perl 还提供了自我判断当前记录级别的能力。虽然 Log::Log4perl 会跳过低级别的日志不输出，但在根据配置文件作出决断之前，仍然要再运行一遍子程序。这简直就是浪费时间，如果预先知道当前的记录级别，跳过不必要的循环输出，则可以省下大量的处理时间。我们可以用 is_debug 方法来确定当前是否在 DEBUG 级别：

```
if ( $logger->is_debug() ) {
  for (@big_array) {
    $logger->debug("big array value: $_");
  }
}
```

每个级别都有与之类似的对应方法，另外 Log::Log4perl 还提供了一些用于识别其他配置状态的方法。

5. 获取更多信息

我们这里仅是对 Log::Log4perl 作了最为简略的介绍，不用担心，更详细的信息可以参阅 Log::Log4perl 的文档，各种常见的应用情况都有对应的例子可以借鉴。如果有兴趣，可以到 SourceForge（http://log4perl.sourceforge.net/）看看 Log4perl 项目的进展。此外，因为配置方式十分相似，所以许多 Log4j 的配置示例也可以借用。

6. 要点

- 用 Log::Log4perl 为程序添加强大的、可配置的日志记录能力。

- 从代码中分离日志记录的配置选项，以获得更佳的灵活性。
- 检查当前记录级别，直接跳过 Log4perl 将会忽略的日志记录。

条款 114　明白循环内的数组何时会被修改

一直以来，许多 Perl 程序员常常因为搞不清楚循环内的数组何时会被修改，而屡屡受挫。让我们简单来说吧，在 for 循环、map 以及 grep 内部，修改特殊变量 $_ 就等同于修改数组内的对应元素。

比如下面这三条语句，都在循环时对原来的数组内容做了修改，每个数字依次自增一：

```
my @array = ( 1 .. 10 );

$_++ for @array;
my @incremented = map { $_++ } @array;
my @grand_plus  = grep { $_++ > 1000 } @array;
```

下列对于使用 for 和 foreach 循环的语句，指定的控制变量，然后直接修改控制变量的值：

```
my @array = 0 .. 5;

foreach my $elem ( @array ) {
  $elem++;
}

print "@array\n"; # 1 2 3 4 5 6
```

为了优化性能，Perl 实际上只是将控制变量当作原始数据的别名而已，并不会在内存中复制相同的内容生成新的变量，所以修改控制变量的值，实际上就是在修改原始数组内对应元素的值。

如果修改的不是数组元素的值，而是数组结构本身，则可能会出现一些有趣的结果。下面的代码遍历十个元素大小的数组，但每次都 pop 一个数组尾部的元素：

```
my $sum = 0;
my @array = ( 1 .. 10 );
foreach (@array) {
  pop @array;
  $sum += $_;
}
print "The sum is $sum\n"; # 结果是 15 而非 55
```

只有开头的五个元素会被计入总和。这是因为每次累加前，数组尾部都会减少一个元素，第一次循环结束后，原来的数组就只包含九个元素了，第二次循环后，便只剩下八个元素。一直到第五次循环结束，原来的数组就剩下开头的五个元素，没有新的元素，整个循环到此结束。

与此类似，循环时也可能动态添加新的元素。下面的例子就是一个死循环，每处理一个元素又新添一个元素，自然无法终结循环：

```
my $max   = 10;
my @array = ( 1 .. $max );
foreach (@array) {
  print "$_ ";
  push @array, ++$max;
}
```


www.TopSage.com

这个循环开始时打印 1，然后是 2，往后就一直下去打印所有的自然数，无穷无尽（或者严格来说，应该是直到内存耗尽为止）。

那如果是添加元素到数组头部呢？

```
my $max   = 10;
my @array = ( 1 .. $max );
for (@array) {
  print "$_ ";
  unshift @array, ++$max;
}
```

现在这个循环会将 1 永远打印下去。看上去好像一直在数组第一个元素的位置上止步不前，就算把 11~15 逐个添加到数组头部也是这样。事实是这个样子吗？如果中途停止追加新元素会发生什么状况？

```
my $max   = 10;
my @array = ( 1 .. $max );
foreach (@array) {
  print "$_ ";
  unshift @array, ++$max if $max <= 15;
}
```

现在它会打印 **1 1 1 1 1 1 2 3 4 5 6 7 8 9 10** 这串数字序列，这一定和你预想的不同吧。

这同 unshift 和 foreach 的内部实现机制有关，由于它们直接在内存中修改原始数组的表示方式，所以按照原来的内存地址循环到下一个元素，就有可能得到意料之外的结果。类似的效果同样发生在 shift 操作移除头部元素的情况，下面的代码会打印 **1 3 5 7 9**。从结果来看，就好像每次移除的是下一个位置上的元素，而非整个数组的第一个元素。

```
my @array = ( 1 .. $max );
for (@array) {
  print "$_ ";
  shift @array;
}
```

要点

- 不要修改循环内的控制变量。
- foreach、map 和 grep 都可以在其内部直接修改原始数组中的内容。
- 在遍历数组时，不要添加或移除该数组内的元素。

条款 115　不要用正则表达式提取逗号分隔的字串

以逗号分隔各项数据的格式，看似容易解析，实则并非如此。开始的时候每个人都会这么想，就好像成人礼一样，人人都会经历一次。而这主要是因为人们还未看到过这种格式的各种具体的变形，所以才会妄加判断。

下面的字串，形式上非常简单，每个字段的值以逗号分隔：

www.TopSage.com

```
my $line = 'Buster,Mimi,Roscoe';
```

只消用 split 断开逗号返回其间数据即可，写出来差不多是这样：

```
my @cats = split /,/, $line;
```

不过要是有人在数据首尾添了空格什么的：

```
my $line = 'Buster, Mimi, Roscoe';
```

问题就来了。空格究竟是字段内容的一部分呢，还是为了格式上便于阅读而额外添加的？如果认为是后一种情况，那原来的 split 代码应该要调整为：

```
my @cats = split /\s*,\s*/, $line;
```

事情开始变得复杂起来。如果某项数据本身用引号括起了若干字段，并且其中也使用了逗号作为分隔符，该怎么办：

```
my $line = '"Bean, Buster", Mimi, Roscoe';
```

已经不像原来那么简单了，对吧？由于没有约定一个格式规范，若某项数据中包含换行符，那最后提取出来的数据就会完全乱套：

```
my $line = '"Bean\nBuster", Mimi, Roscoe';
```

又或者某项数据内包含双引号，而转义时恰恰使用两个双引号的形式，又该怎么办：

```
my $line = '"""Bean"", Buster", Mimi, Roscoe';
```

几乎所有尝试写出来的正则表达式都不能完备地处理这种 CSV 格式的数据。与其纠结于此，浪费大量时间调整正则表达式，不如直接用 Text::CSV_XS 模块处理。它肯定会比我们写的更快，用起来也更容易：

```
my $csv = Text::CSV_XS->new( { binary => 1 } )
  or die 'Cannot use CSV: ' . Text::CSV->error_diag;

open my $fh, '<', $file or die "$file: $!";

while ( my $row = $csv->getline($fh) ) {
  next unless $row->[2] =~ m/$pattern/;
  push @rows, $row;
}

$csv->eof or $csv->error_diag;
close $fh;
```

要点

- 不要用正则表达式解析逗号分隔的字串。
- 用 Text::CSV_XS 模块来处理 CSV 格式的数据。

条款 116　用 unpack 处理固定列宽的数据

并非每次都得用正则表达式或 split 函数提取文本的各列数据，如果数据以固定列宽的形式

www.TopSage.com

排列，则可以方便地用 unpack 提取至一组对应变量。

1. 利用 unpack 解析

看下面这段文本，想一想如果用正则表达式该怎么提取各个字段：

```
ID  First Name  Middle  Last Name
 1  brian       d       foy
 2  Joshua              McAdams
 3  Joseph      N       Hall
```

像这样的情形，利用空白字符来做分隔的 split 恐怕已经无能为力，因为该数据是按字段的宽度排布的，而不是由空白字符的类型或数量来决定它们的间隔位置。特别是当某些字段没有内容时，像上面没有中间名的那行，空白字符分隔数据的方式肯定会出乱子。

通过观察我们可以发现，数据的格式是固定的。ID 列总是占据第 1 列和第 2 列，名字总是占据第 4 列到第 15 列，依次类推。

于是我们可以构造一个 pack 能够理解的格式定义：

```
my $format = 'A2 @4 A10 @16 A6 @24 A*';
```

@字符并非代表某个特定字段，它只是用于告诉 unpack 移至指定的绝对位置，然后再继续处理下一部分的格式。而这里的 A 字符会提取对应长度的数据，并自动去掉尾部的空白字符。

由于字段名和字段内容都以相同的格式排布，所以用上面的格式定义可以一并处理：

```
my $string = <<'COLUMNAR';
ID  First Name  Middle  Last Name
 1  brian       d       foy
 2  Joshua              McAdams
 3  Joseph      N       Hall
COLUMNAR

open my ( $fh ), '<', \$string;

my $format = 'A2 @4 A10 @16 A6 @24 A*';

my @headers = unpack $format, <$fh>;

my @names; # 准备一个数组，存放表示各行数据的散列引用

while( <$fh> ) {
  my %hash;
  @hash{ @headers } = unpack $format, $_;
  push @names, \%hash;
}

use Data::Dumper::Names;
print Dumper( \@names );
```

打印出来的结果，显示了由 unpack 展开的各项字段内容。请注意，其中尾部的空白字符已经被自动去除：

```
@names = (
```

www.TopSage.com

TopSage.com

```
    {
      'First Name' => 'brian',
      'ID'         => ' 1',
      'Middle'     => 'd',
      'Last Name'  => 'foy'
    },
    {
      'First Name' => 'Joshua',
      'ID'         => ' 2',
      'Middle'     => '',
      'Last Name'  => 'McAdams'
    },
    {
      'First Name' => 'Joseph',
      'ID'         => ' 3',
      'Middle'     => 'N',
      'Last Name'  => 'Hall'
    }
);
```

当然，这里唯一比较麻烦的，就是构造格式定义时对列宽长度的估计，用以以设定不同字段的@位置。有个小窍门能让这项工作变得稍微容易些，先大致猜测一下偏移位置并运行 unpack 看输出结果，然后一点点调整到结果正确为止。

2. 要点

- 不要用正则表达式或 split 解析固定长度的数据。
- 用 unpack 解析固定列宽的数据才是正道。
- unpack 时用 A 字符，可以自动去除对应数据尾部的空白字符。

条款 117　用 pack 和 unpack 对数据作变形处理

Perl 内建的 pack 和 unpack 函数，可以比作是“瑞士军刀”上最强大锋利的刀刃了。也许在创造这两个函数之初，人们只是想要完成一些无趣的，二进制数据和 Perl 数据之间的相互转换工作，像字符串、整数等，不过，pack 和 unpack 还可以用来做许多其他更有意思的事情，有些应用甚至称得上是标新立异。想要查看更多的用法示例，还请阅读 **perlpacktut** 文档。

1. 打包数据

pack 的工作方式看起来多少有点和 sprintf 相像。它根据格式定义字串，对给定的一组数据进行格式化，然后返回拼接而成的字符串：

```
# 返回的字符串为 Perl，由四个无符号字符类型的数字拼接而成
my $packed_chars = pack( "CCCC", 80, 101, 114, 108 );
```

而 unpack 的工作方式恰恰相反：

```
my @ints = unpack( "CCCC", "Perl" ) # 得到数组(80, 101, 114, 108);
```

用于定义格式的字串，由一系列单个字母的指示符组成，每个指示符都对应于打包或解包的

www.TopSage.com

数据类型。表 13-1 列选了部分常用的指示符（完整列表请参考 **perlfunc** 文档中关于 pack 函数的部分。）：

表13-1　常用的pack格式指示符

指示符	描述
a	任意长度的二进制数据字串，若长度不够会自动在尾部填充null字符
A	文本（ASCII）字串，若长度不够会自动在尾部填充空格字符
h	十六进制字串（字节低四位靠前）
H	十六进制字串（字节高四位靠前）
c	有符号字符值（8比特）
C	无符号字符值（8比特）
s	有符号短整型值（16比特）
S	无符号短整型值（16比特）
l	有符号长整型值（32比特）
L	无符号长整型值（32比特）
n	无符号短整型值（16比特）以“网络字节序”（大字节序）表示
N	无符号长整型值（32比特）以“网络字节序”（大字节序）表示
v	无符号短整型值（16比特）以“VAX字节序”（小字节序）表示
V	无符号长整型值（32比特）以“VAX字节序”（小字节序）表示
u	uuencod编码的字串
U	Unicode字符的内部编码数字。字符模式下是对字符本身进行编码，字节模式下以UTF-8（或者在EBCDIC平台上以UTF-EBCDIC）方式编码
@	指定当前指示符对应的绝对位置

每个指示符都可以跟一个表示重复使用次数的数字，以表明后续多少单位的数据将按此格式处理。而表示字符数据的指示符（A，a，B，b，H 和 h）对应的重复单位各不相同，每两个一组分别按字节（byte）/位（bit）/四字节（nybble）来计数。如果是星号的话，就说明不限重复次数，直到末尾。

unpack 函数还可以用来计算数据的校验和（checksum），只需在指示符前面加上百分号和以多少位求校验和的数字即可。每个字符计算得到的校验和会再求和校验一次，最终得到一个独立的校验结果：

```
unpack "c4",    "\1\2\3\4"; # 四个数字：1, 2, 3, 4
unpack "%16c4", "\1\2\3\4"; # 求和再按 16 位（即 2**16）求余，得到 10
unpack "%3c4", " \1\2\3\4"; # 求和再按 3 位（即 2**3）求余，得到 2
```

2. 借助 pack 来排序

假设有一组以数字表示的因特网地址（即写成字符串的形式）需要排序，比如下面这些：

```
11.22.33.44
1.3.5.7
23.34.45.56
```

排序的结果应该按照“数字”值而非字面上的字符串。也就是说，应该先按数字大小对第一

www.TopSage.com

列进行排序，如果数字相同，再继续依次按第二列、第三列以及第四列的数字顺序排序。如果一如平常，直接用 sort 函数来做的话，结果将会是错误的（见条款 22）。即使改为按数字排序也一样不对，因为 sort 只会关心第一个位置上的数字。而利用 pack 函数则可以很好地解决这样的问题：

```
my @sorted_addr =
  sort {
    pack( 'C*', split /\./, $a ) cmp
    pack( 'C*', split /\./, $b )
  } @addr;
```

不过考虑到性能问题，上面的代码完全可以改写成下面这样的施瓦茨变换形式（见条款 22）：

```
my @sorted_addr =
  map  { $_->[0] }
  sort { $a->[1] cmp $b->[1] }
  map  { [ $_, pack( 'C*', split /\./ ) ] } @addr;
```

请注意，在 sort 内部做比较的运算符是 cmp 而非 <=>。pack 函数会将一系列数字（比如 11，22，33，44）转换为一个四个字节长度的字符串（\x0b\x16\x21\x2c），而按此字符串排序的结果，恰恰就是我们期望得到的。当然，对于这个例子，我们还可以用 Socket 模块提供的函数作转换，如下所示：

```
my @sorted_addr =
  map  { $_->[0] }
  sort { $a->[1] cmp $b->[1] }
  map  { [ $_, inet_aton($_) ] } @addr;
```

3. 处理十六进制转义序列

由于 pack 和 unpack 都能处理十六进制字符串，所以常被用来构造十六进制的转义序列或是将它们转换回来。

比如说，在 Web 编程时经常会碰到对 URI 字符串的转义处理，把编码过的 URI 转回到提交前的原始字符串。转义过程实际很简单，将每一个转义形式的字符串——一个百分号，后跟两个十六进制数字——全部替换为以此十六进制表示的字符。像 a%5eb 解除转义后应该得到 a^b。按照这个思路，可以直接写一条 Perl 的替换正则表达式：

```
$_ = "a%5eb";
s/%([0-9a-fA-F]{2})/pack("c",hex($1))/ge;
```

这段代码总是出现在陈旧的手写 CGI 脚本程序中，碰到特殊情况也许还会出错。不过，既然是人人都需要的功能，好多常见的任务和工作也都需要做这样的处理，何不采用现成的专为此设计的模块：

```
use URI::Escape;
$_ = uri_unescape "a%5eb";
```

4. 处理字节序

计算机内部对于多字节数据的存储，按字节顺序的先后不同，分为两种不同架构。一种是大

www.TopSage.com

字节序（big-endian），对数值影响最大的字节位于低位地址，另一种是小字节序（little-endian），对数值影响最小的字节位于低位地址。

还是来看看实际的十六进制数字 0xAABBCCDD 吧。在大字节序中，保存在内存中由低到高的位置上的，分别是 AA、BB、CC 和 DD，权重最大的数字先出现。而在小字节序中，内存由低到高的分别是 DD、CC、BB 和 AA，权重最大的数字最后才出现。网上关于字节序的概念随处可见，此处从略。

有关字节序的问题，最常见于读取二进制文件的操作。请看下面的代码，读入四个字节后，须以正确的字节序还原为原始数据。如果在 unpack 中按当前计算机的字节序以 L 格式展开，可能得到的是错误的内容：

```
read $fh, $buffer, 4;
my $number = unpack 'L', $buffer; # ???
```

不过，对于现有的文件，我们多半已经有对它所使用的字节序有所了解，那就直接告诉 unpack 该用什么方式展开好了。

比如我们知道原始数据是以小字节序存储的，则可以用 V（表示按“VAX”机的顺序）返回 32 比特的无符号整形数字，就算我们用的计算机是大字节序也没问题：

```
read $fh, $buffer, 4;
my $number = unpack 'V', $buffer; # 小字节序
```

如果原始数据是以大字节序存储的，则可以用 N（表示按网络“network”顺序）：

```
read $fh, $buffer, 4;
my $number = unpack 'N', $buffer; # 大字节序
```

对比看一下，将数字 0xAABBCCDD 以不同字节序打包，然后看看得到的是怎样的字节序列：

```
my $ddccbbaa = pack 'V', 0xAA_BB_CC_DD; # 小字节序
my $aabbccdd = pack 'N', 0xAA_BB_CC_DD; # 大字节序

printf "\$aabbccdd is 0x%s\n",
  join "_",
  map { sprintf '%X', ord }
  split //, $aabbccdd;

printf "\$ddccbbaa is 0x%s\n",
  join "_",
  map { sprintf '%X', ord }
  split //, $ddccbbaa;
```

即便打包的是同一个数字，也可以看到实际存储到内部的，是顺序完全不同的数据：

```
$aabbccdd is 0xAA_BB_CC_DD
$ddccbbaa is 0xDD_CC_BB_AA
```

仅看数字本身是无法判断它究竟属于何种字节序的，除非我们预先就已经知道。当解包方式不同于打包方式时，得到的必然是错误的数字：

```
my $packed = pack 'L', 137;

printf '%d', unpack 'N', $packed; # -1996488704
```

www.TopSage.com

我们必须决定该用何种方式的 unpack 来得到正确的数字。一般来说，得先查看下数据格式的定义，或者直接问知道的人。只有明确知道原始的字节序，才能以相同方式返回原始数据。正是基于这样的原因，许多计算机系统都约定使用网络字节序，以保证两边都能理解数据内容。

不过有时候，通过一小段已知数字的字节序列，数据本身也能提供一些线索，比如 Unicode 编码的文件会有一个字节序（byte-order）标记 U+FEFF，籍此可以判断它是以何种字节序保存的。尽管实际使用的字节序列和文件中的保存方式完全一致，但一部小字节序的计算机，还是会把 U+FEFF 转换为数字 0xFFFE。

照着这个思路实际做个实验，创建一个包含该字节序列的“文件”。打开一个指向某个字符串的文件句柄（见条款 54），读入两个字节，然后以 S（16 比特无符号短整形数字）格式解包。要判断当前计算机的字节序，只需比较 unpack 得到的结果和不同字节序机器上的表示即可：

```
my $string = "\xFE\xFF";
open my($fh), '<', \$string;

read $fh, my( $bom ), 2;
my $unpacked = unpack 'S', $bom;

if( 0xFEFF == $unpacked ) {
  print "Big Endian!\n"
}
elsif( 0xFFFE == $unpacked ) {
  print "Little Endian!\n"
}
else {
  print "What the heck are you?\n"
}
```

5. Uuencode 编码

Uuencode 编码是一种将二进制数据转换为 ASCII 码字符表示的编码方式，这样数据就能顺利通过 7 位的通道传输，比如旧时的电子邮件系统，从而保证大于 7 位的数据不会在传输过程中遭到破坏。如果长久以来你一直在探寻如何避免邮件服务器弄乱重音字符，现在不必为此烦心了，因为用 Perl 实现非常简单。单是 pack 函数的 u 格式就能搞定：

```
use utf8;

my $string = <<"HERE";
Can my fiancée, Ms. Sørenson, send you her résumé?
HERE

my $uuencoded = pack 'u*', $string;

print "=begin 644 $filename\n", $uuencoded, "`\n=end\n";
```

将 uuencode 编码过的文本转换回原始的文本同样简单。假设编码后的数据保存在某个文件中，先读取文件内容，去掉编码文字的标识头和结束标记，然后用 unpack 解开来：

```
use utf8;
```


www.TopSage.com

大家网 TopSage.com

```
while (<>) {
  last
    if ( $mode, $filename ) = /^begin\s+(\d+)\s+(\S+)/i;
}

if ($mode) {
  open my ($fh), '>:utf8', $filename
    or die "couldn't open $filename: $!\n"
  chmod oct($mode), $filename
    or die "couldn't set mode: $!\n";

  print "$mode $filename\n";

  while (<>) {
    last if (/^(`|end)/i);
    print $fh unpack( 'u*', $_ );
  }
}
```

6. 要点

- 利用 pack 可以将一系列数据打包成单一字符串。
- 恢复数据时可以使用 unpack 函数。
- 用恰当的 pack 格式打包数据。

条款118　借用 typeglob 访问符号表

老实说，我们完全没期望你对 Perl 里面的符号表或者 typeglob 有什么深入了解，那不过是 Perl 内部用来追踪包变量、已命名的子程序定义以及裸字文件句柄的一种方式。多半情况下我们都可以借助词法变量、引用或面向对象的编程方式来操作，而不必同符号表打交道。但偶尔也会有不可避免的时候。比如老旧的 Perl 代码中就经常出现一些操作符号表以及使用 typeglob 的地方，所以对此有所了解，明白它们究竟在做些什么，还是非常有必要的。

在 Perl 里面，同一个标识符可用于表示不同的变量——比如 $foo、@foo、%foo、&foo、foo（作为文件句柄）等，Perl 在内部符号表中都以“foo”作为入口。

我们可以直接操作符号表中各项变量的内容，这得用到一个称为 **typeglob** 的结构。所谓的 typeglob，就是“各种类型的总体”（a glob of types）的意思，在标识符前加注星号——比如 *foo，就表示所有以该名字注册在符号表中的各式变量。

所以，借助 typeglob 可以对所有类型的同名变量构造一组别名：

```
*ren = *stimpy; # 现在，$ren 成了 $stimpy 的别名，@ren 成了 @stimpy 的别名，其余类同

# 同上，但显式声明符号表的包名
*main::ren = *main::stimpy;
```

也可以在有限词法作用域范围内使用 typeglob，所生成的别名仅在该范围内有效：

```
$ren = 'hello stimpy';
```

www.TopSage.com

```
{
  local *ren = *stimpy; # $ren、@ren 等仅在此范围内有效
  $stimpy = 'yello ren';
  print "$ren\n";        # 打印'yello ren'
}

print "$ren\n";          # 仍然会打印'yello ren'
$stimpy = 'later skater';
print "$ren\n";          # 还是会打印'yello ren'
```

另外，我们还可以通过标识符文字直接修改符号表散列，生成新的别名（但这和之前的软引用方式不同）：

```
$main::{'ren'} = $main::{'stimpy'};
local $main::{'ren'} = $main::{'stimpy'};
```

typeglob 也可以作为参数，传递给子程序，或是保存在标量变量内：

```
my @g = ( *foo, *bar ); # 将 typeglob 存入数组。但必须是包变量
our ( $foo, $bar ) = ( "ren", "stimpy" );

*s = $g[0];             # 使用它们
*t = $g[1];             # 或者改用更简捷的写法：(*s, *t) = @g
print "$s and $t\n";    # 打印“ren and stimpy”
```

单是某种类型的变量，比如某个数组或某个子程序，也能单独生成对应的别名，只需将它们的引用赋值给 typeglob 就行。像下面的代码，把 hello 映射为子程序 world 的别名[①]：

```
sub world { "world\n" }
*hello = \&world;

my $hello = "hello";
print $hello . ", " . &hello; # 打印“hello, world”
```

如果要临时替换某个子程序的定义，便可利用这种技术，在本地词法作用域范围内借助 typeglob 生成新的子程序定义。Perl 会明白当前该调用的是哪个子程序：

```
sub greet { print "Hello!\n" }

greet();

{
  local *greet;

  *greet = sub { print "How you doing?\n" };

  greet();
}

greet(); # 恢复到原来
```

我们还可以用 typeglob 来对文件句柄和目录句柄（见条款 52）做局部化，像这样：

① 但只是修改了 hello 作为子程序时的引用，不影响后面作为标量变量的$hello 的值。——译者注

www.TopSage.com

```
sub some_file_thing {
  local *FH; # FH是该子程序内部的表示符
  open FH, "foo";
  ...;
}
```

在平时使用引用的地方，我们也可以直接替换为typeglob的写法（若非必要，请不要这么写）：

```
sub you { print "yo, world\n" }
&{*you}();  # 打印“yo, world”
```

Perl还有一个称为“typeglob subscript”的语法，*FOO{BAR}，由此可以展开对应类型变量的引用：

```
$a    = "testing";
@a    = 1 .. 3;
$sref = *a{SCALAR};
$aref = *a{ARRAY};
print "$$sref @$aref\n";
```

要点

- Perl使用符号表跟踪所有的包变量。
- 可以使用typeglob生成变量的别名。
- 需要重新定义子程序时，可以借助typeglob修改该子程序在符号表中的引用。

条款119　用BEGIN初始化，以END善后

Perl提供了一种特别的语法，BEGIN块，用于在程序开始运行前执行初始化代码。与此对应，Perl还提供了END块语法，在程序退出运行前执行善后代码[①]。

1. BEGIN块

BEGIN块内的代码，会在自身编译完成后第一时间立即执行，然后继续编译和运行后续的程序主体代码。比如，用BEGIN块预先定义好某个子程序将要用到的变量，并将它作为子程序的私有词法变量，得在子程序使用它之前先定义好并初始化：

```
BEGIN {
  my $dow = qw(Sun Mon Tue Wed Thu Fri Sat);

  sub dow {
    $dow[ $_[0] % 7 ];
  }
}
```

有时，为了临时修复某个模块的bug，也会用到BEGIN块。先期替换掉原模块中的部分代码，以保证后续程序运行正常：

① 实际上，Perl沿袭了awk命令的BEGIN/END代码块的写法。——译者注

www.TopSage.com

```
BEGIN {
  use Some::Module;
  no warnings 'redefine';
  *Some::Module::some_sub = sub { ... 修正后的代码 ... }
}
```

有时，需要在程序运行前检查某些运行环境的状况，比如要求多线程运行的程序，运行前最好先检查下当前 perl 是否支持多线程：

```
BEGIN {
  use Config;

  die "You need a threaded perl to run this!"
    unless $Config{useithreads} eq 'define';
}
```

更有趣的情况，如程序需要建立数据连接，有时候我们仅仅只希望运行 perl -c 做语法检查，此时我们就不希望连接数据库，可以利用$^C 这个特殊变量判断当前是否打开了-c 开关，然后分流处理：

```
BEGIN {
  if ($^C) {
    print "I'm just checking my syntax with -c\n";
  }
  else {
    print "I'm getting ready to run, so I should "
      . "get ready\n";
    my $dbh = DBI->connect(...);
  }
}
```

可能我们并不觉得有必要检查-c 开关，不过要是我们的编辑器或者 IDE 自动编译测试代码时不断显示程序出错信息呢？这多半是因为它们幕后使用 perl -c 作语法检查。

BEGIN 块可以不止一个，执行时按定义的先后顺序依次执行。稍后我们就会看到具体示例。

2. END 块

END 块内的代码会在 Perl 程序正常终止前执行。对于善后清理工作来说，END 块不可或缺，比如撤销文件锁、释放资源等诸如此类：

```
END {
  my $program_name = basename($0);
  unlink glob "/tmp/$program_name.*";
}
```

END 块也可以有多个，如在程序“既定”的中止时刻运行——即脚本运行到末尾，或者 exit 命令退出，或者 die 命令中止运行等。但多个 END 块是按照编译时相反的顺序依次执行的，也就是说，最后一个先执行，第一个最后执行。请看下面的代码片段，实际运行顺序同其输出内容，先依次执行各个 BEGIN 块，然后运行程序主体代码，最后逆序运行每个 END 块。

```
BEGIN { print "1. I'm first\n"; }
END   { print "6. I'm sixth\n" }
```

www.TopSage.com

```
print "4. I'm fourth\n";
BEGIN { print "2. I'm second\n"; }
BEGIN { print "3. I'm third\n"; }
END   { print "5. I'm fifth\n" }
```

但请注意，如果程序非正常退出，比如收到系统发来的信号，程序将直接结束运行，而不会有机会运行 END 块[①]。

3. 要点

- 用 BEGIN 块在编译时先运行代码。
- 用 END 块在程序结束运行时执行善后代码。

条款120　用单行 Perl 命令作为迷你程序

Perl 历来都和系统管理工作密不可分，人们总喜欢写单行命令当作迷你程序使用。**perlrun** 文档列出了所有命令行参数选项，借助这些参数就能写出各式迷你程序，但这里只介绍一些最常用的。

因为只是一行程序，许多概念都被淡化到几乎不用，像词法变量、描述性名称以及代码作用域等在短小的程序中意义就不大。而默认变量的使用（见条款 15 及条款 16），又非常适合用在命令行以节省输入。但要注意适度，既不能为求短小而忽略程序的可读性，也不能写得太长造成阅读理解上的困难。

1. -e 开关

要在命令行运行一小段 Perl 代码，得使用 -e 开关，或者 Perl 5.10 版本以上的话，可以用 -E 开关打开其他可选特性。而跟在 -e 开关后面的一段字串，就会作为 Perl 程序的源代码来执行。因为定义这段字串本身就需要使用引号，所以在程序内部表示字符串时，得改用泛引用的方式（见条款 21），这样就不致于让 shell 误会命令行的参数内容（像 Windows 的 cmd 程序就要求参数必须以双引号引起来）：

```
% perl -e "print qq(Hello World\n)"

% perl -E "say q(Hello World)"
```

可以使用 -l 开关对输出内容自动添加换行符，这就免去了专门为了加一个换行符而特意使用双引号：

```
% perl -le "print q(Hello World)"

% perl -le "print q(The time is ), scalar localtime"
```

所以，要每行打印一条 @INC 中包含的模块搜索路径，只需要这么写：

```
% perl -le 'print for @INC'
```

① 这是因为控制权不在 Perl 程序本身，而是由更高级别的系统进程管理。——译者注

www.TopSage.com

要给这样的小程序传递参数也没问题。我们知道，在子程序以外的地方使用 shift，意思就是将@ARGV 中的参数提取出来（见条款 16）。下面一行是将十进制数字转换为十六进制表示的单行程序：

```
% perl -e "printf qq|%X\n|, int( shift )" 137
93
```

单行程序调试妥当之后，如果经常使用，可以在 shell 中为它设置别名，以后只需输入别名，就会直接运行这条单行程序。如果经常要做各种进制间的转换，不妨设立多条别名，比如二进制（以 b 表示）、八进制（以 o 表示）、十进制（以 d 表示）、十六进制（以 h 表示）之间的两两互换：

```
alias d2o="perl -e 'printf qq|%o\n|, int( shift )'"
alias d2b="perl -e 'printf qq|%b\n|, int( shift )'"
alias h2d="perl -e 'printf qq|%d\n|, hex( shift )'"
alias h2o="perl -e 'printf qq|%o\n|, hex( shift )'"
alias h2b="perl -e 'printf qq|%b\n|, hex( shift )'"
alias o2h="perl -e 'printf qq|%X\n|, oct( shift )'"
alias o2d="perl -e 'printf qq|%d\n|, oct( shift )'"
alias o2b="perl -e 'printf qq|%b\n|, oct( shift )'"
```

系统中经常以纪元秒（epoch）表示的时间，如果要换算成直观可理解的时间字串，可以用下面这条别名：

```
alias e2t="perl -le 'print localtime( shift )'"
```

2. -n 开关

当使用-e 指定一段要执行的代码时，再指定 -n 便会让这段程序封装成 while 循环内的代码运行。每读入文件一行，便作为 while 循环获取的输入，然后依次处理：

```
% perl -ne "print" fileA fileB
```

所以上面这段代码，完全等效于下面这段程序，-e 指定的部分，就是下面 while 循环内的代码：

```
while (<>) {
  print;
}
```

这就让我们可以将输入的每一行做一定修改后重新依次输出。比如要为每行文本添加行号，只需在输出时包含特殊变量 $. 即可：

```
% perl -ne 'print qq($.: $_)' fileA
```

请注意，这条单行程序中我们使用了单引号，否则 UNIX 的 shell 会把双引号中 $ 开头的变量当作 shell 的变量加以解释，但这里的 $. 和 $_ 都是 Perl 自己的特殊变量。如果是在 Windows 上倒是可以用双引号，但 UNIX 上则必须对 $ 进行转义，所以一定要用双引号的话，代码会看起来很乱：

```
% perl -ne "print qq(\$.: \$_)" fileA
```

如果只想输出其中连续的几行内容，倒也不难。范围操作符（即 .. ，称作 flip-flop）在标

www.TopSage.com

量上下文环境时，如果输入值位于两侧取值范围之外，便返回假，否则返回真。而当仅仅给定一个数字范围时，它会默认拿表示行计数的 $. 来作比较。所以下面这条单行程序会打印第四行至第七行的内容：

```
% perl -ne 'print qq($.: $_) if 4 .. 7' fileA
```

如果要输出单数行，一样简单。借助求余操作符 % 即可实现：

```
% perl -ne 'print qq($.: $_) if $. % 2' fileA
```

由于-n 是将代码简单封装到 while(<>){和}之间的，所以偶尔还能利用它要点小花招。-n 会在-e 指定的代码末尾添加一个虚拟的花括号作为结束，而有些代码我们并不希望放在循环体内执行，可以利用 END 块（见条款 119），有意构造一个循环体外的代码段，像这样：

```
% perl -nle '$count++ } END { print $count' *.pl
```

所以上面这条单行程序，展开来的话就等同于下面这段代码，虽然格式上看起来很怪：

```
while( <> ) {
    $count++ } END { print $count
}
```

类似的，有时我们也希望在循环开始之前先执行某些代码，利用 BEGIN 块（见条款 119）好了：

```
% perl -nle '$count++ } \
     BEGIN { print q(Counting ) . @ARGV . q( files) } \
     END { print $count' *.pl
```

3. -p 开关

-p 开关和-n 类似，但会在每次循环结束前自动打印 $_ 的值。

```
% perl -pe 's/buster/Buster/g' fileA fileB
```

-p 一样会把程序变为 while 循环，将给定的代码作为循环主体执行，结束循环前，添加一行打印当前默认变量 $_ 内容的语句：

```
while (<>) {
  s/buster/Buster/g;
  print;
}
```

比如要把文件中每行末尾的换行符转成 UNIX 标准的换行符，只需关注替换操作本身，之后的打印输出就交给-p 完成好了：

```
% perl -pe 's/\012?\015/\n/g' file-dox.txt > \
    file-unix.txt
```

4. -i 开关

-i 开关会施展魔法，让你对文件能有直接作内嵌的。实际上使用 -i 时，幕后 Perl 会先重命名该原始文件，打开它读入每一行内容，然后写入到名字为原文件名的新文件中。这样，要对文件内容逐行转换，也只需要关注转换逻辑的代码，其他事项交给 -i 处理。像下面这条单行程序，

www.TopSage.com

就是利用 -pi 将文件中所有的冒号转换为制表符：

```
% perl -pi -e 's/:/\t/' fileA fileB
```

-i 开关会直接拿原始数据来处理，所以默认情况下 perl 并不会保留备份文件。但如果担心代码运行有误，想要留存备份以避免损失，可以在 -i 开关后指定备份文件的后缀名称，比如 .old。现在 perl 会把原始数据存为备份文件 fileA.old 和 fileB.old。

```
% perl -pi.old -e 's/:/\t/' fileA fileB
```

要将文件中出现的所有数字都替换为自增一的数字，可以使用全局替换：

```
% perl -pi.bak -e 's/(\d+)/ 1 + $1 /ge' fileA
```

也可以只替换那些不与任何单词相邻的独立数字：

```
% perl -pi.bak -e 's/\b(\d+)\b/ 1 + $1 /ge' fileA
```

或者将制表符展开为等同长度的空格字符。这个做起来得动动脑筋，因为每个制表符会根据上下文自动显示为不同宽度，所以不能简单替换所有制表符为固定长度的空格序列。不过我们可以想办法侦测当前制表符所在位置，然后计算对应显示的空格个数：

```
% perl -pi -e 's/\t/ q( ) x (4 - pos() % 4) /ge' tabs.txt
```

使用-0 开关，可以指定读取输入时以何为界分割每条输入内容，可以用八进制或是十六进制格式的值。如果指定 00，则打开段落模式。比如，下面的例子重新规排段落行宽。它先将所有连续的空白字符替换为单个空格，然后查找单个空格及之前 73 个字符的组合字串，在其末尾追加换行符。最终输出 $_ 前，再追加两个换行符以切分出一个新的段落：

```
% perl -000 -pi.bak -e \
    's/\s+/ /g; s/(.{50,73})\s/$1\n/g; $_.=qq(\n\n)'
```

其实这段程序是使用 \K 的绝佳例子，它会让跟在之前的模式所匹配的内容在替换时被忽略，即那段内容不会被替换掉：

```
% perl5.10.1 -000 -pe \
    's/\s+/ /g; s/(.{50,73})\K\s/\n/g; $_.=qq(\n\n)'
```

5. -M 开关

在命令行要使用某个模块的功能，可以用 -M 开关先加载该模块：

```
% perl -MLWP::Simple -e "getprint 'http://www.example.com'"

% perl -MFile::Spec::Functions -le \
    'print catfile( @ARGV )' a b c
```

加载模块的同时要引入它的符号项目，可以在模块名后添加等号并列出符号名称：

```
% perl -MList::Util=shuffle -le \
    "print for( shuffle(@ARGV) )" a b c
```

要加载多个模块，只需多写几个 -M 指定就好了：

```
% perl -MList::Util=shuffle -MList::MoreUtils=uniq \
    -le "print for( shuffle( uniq @ARGV) )" a b c a h g
```


www.TopSage.com

6. -a 和-F 开关

-a 开关会将每行内容按空白字符切分后存入特殊变量 @F 里面。和之前见过的 END 块拼补技巧类似，这次我们可以用 for 来做后续处理，得到词频统计：

```
% perl -anle '$S{$_}++ for @F } \
    for( keys %S ) { print qq($_ $S{$_})'
```

进一步，如果要按照词频数量排序，也一样简单：

```
% perl -anle '$S{$_}++ for @F } \
    for( sort { $S{$b} <=> $S{$a} } keys %S ) \
    { print qq($_: $S{$_})'
```

对于以空格字符分隔的单词，上面这种做法正好够用，但假若以其他字符分隔呢？比如冒号？好吧，-F 开关来了，它可以指定切分模式的分隔字符[①]：

```
% perl -aF: -nle '$S{$_}++ for @F } \
    for( sort { $S{$b} <=> $S{$a} } keys %S ) \
    { print qq($_: $S{$_})'
```

如果只需要统计开头的几个单词，比如每行第一到第三个单词，该如何做？使用 @F 的切片形式就可以了：

```
% perl -aF: -nle '$S{$_}++ for @F[0..2] } \
    for( sort { $S{$b} <=> $S{$a} } keys %S ) \
    { print qq($_: $S{$_})'
```

7. 要点

- 可以在命令行直接编写短小精悍的实用小程序。
- 对文件内容做修改时，记得备份原始数据。
- 查阅 **perlrun** 文档学习其他 perl 开关参数的使用。

① 这其实和 awk 是一样的，也是用大写的 F 指定分隔字符。——译者注

www.TopSage.com

附录

Perl 资源

这只是一本关于 Perl 的小册子，其中主要是我们精心筛选出的对 Perl 中级程序员来说最有价值的话题。本书的第 1 版出版于 Perl 的应用规模和社区都比较小的时候，现在两者都有了长足的发展，所以可用的资源也越来越多了。

书籍

下面是我们推荐的书目，不多，但你可以在 http://books.perl.org/找到更多书籍。

- 《Perl 语言入门（第 5 版）》（Sebastopol，CA：O'Reilly Media，2008），作者是 Randal L. Schwartz、Tom Phoenix 和 brian d foy。这是学习 Perl 的经典书籍，涵盖了我们日常编程中最常用的 80%的 Perl 知识。
- *Intermediate Perl*（Sebastopol，CA：O'Reilly Media，2006），作者是 Randal L. Schwartz、Tom Phoenix 和 brian d foy。在学习了《Perl 入门》之后，你应该可以开始深入学习关于引用、包和模块的知识了。
- 《精通 Perl》（Sebastopol，CA：O'Reilly Media，2007），作者是 brian d foy。这本书的主旨不是要教我们更多 Perl 语法，而是如何更加有智慧的使用 Perl 来构建强壮的企业级应用。
- *Higher Order Perl:Transforming Programs with Programs*（San Francisco，CA：Morgan Kaufmann，2005），作者是 Mark Jason Dominus。如果你迫切需要 Perl 的动态子程序带来的能力，可以试着解读这位 Perl 高手设下的迷局。另外你还可以在 http://hop.perl.plover.com/下载此书的免费版本。
- *Object Oriented Perl:A Comprehensive Guide to Concepts and Programming Techniques*（Greenwich，CT：Manning Publications，2000），作者是 Damian Conway。在这本书中我们可以学到的不只是 Perl 的面向对象语法，还有很多理论和面向对象方面的哲理。
- 《Perl 网络编程》（Boston，MA：Addison-Wesley Professional，2001），作者是 Lincoln Stein。尽管这本书面世已经有些日子了，但网络编程最基础的那部分仍然没有发生太大变化。如果你希望理解 Perl 网络编程方面的模块是如何实现的，请先阅读这本书。

www.TopSage.com

大家网 TopSage.com

- 《Perl Testing程序高手秘笈》（Sebastopol，CA：O'Reilly Media，2005），作者是 Ian Langworth 和 chromatic。这本书专注于 Perl 测试方面的最佳实践。
- *Pro Perl Debugging*（New York，NY：Apress，LLC，2005），作者是 Foley 和 Andy Lester。如果你想要知道如何使用 Perl 内置的调试器，这本书能让你了解许多东西，其中有些可能你从未想过可以做到。
- *Writing Perl Modules for CPAN*（New York，NY：Apress，LLC，2002），作者是 Sam Tregar。如果你刚刚开始为 CPAN 编写模块，那么这是一本非常适合的指南，可以在 Apress 公司的站点免费下载，地址是 http://www.apress.com/book/ view/159059018X。
- *The Perl Cookbook, Second Edition*（Sebastopol, CA：O'Reilly Media，2003），作者是 Tom Christiansen 和 Nat Torkington。此书与本书非常类似，其中有很多指导你解决某类实际问题的小技巧。
- *Automating System Administration with Perl:Tools to Make You More Efficient*（Sebastopol，CA：O'Reilly Media，2009），作者是 David N. Blank-Edelman。
- 《Perl Hacks：100 个业界最尖端的技巧和工具》（Sebastopol，CA：O'Reilly Media，2006），作者是 chromatic、Damian Conway 和 Curtis "Ovid" Poe。它能带你学会一些很酷的 Perl 秘技，所以也可以看看这本书，哪怕只是为了找乐子。

Web 站点

- http://perldoc.perl.org/
 在线 Perl 文档。
- http://search.cpan.org/
 了解任何 Perl 模块的所有细节。
- http://learn.perl.org/
 Perl 入门阶段的资源站点。
- http://perltraining.com.au/tips/
 Perl Training Australia 的一些技术分享。
- http://pause.perl.org/
 Perl 作者上传服务器，从此开始你的 CPAN 作者生涯。
- http://www.yapc.org/和 http://yapceurope.org/
 关于 Perl 活动和会议的站点。
- http://www.pm.org/
 在此可以发现你周围的 Perl 用户组。
- http://www.theperlreview.com/
 在 *The Perl Review* 杂志上发布的文章和其他 Perl 资源。

www.TopSage.com

博客和播客

关于 Perl 的博客很多，大多数都在一些 blog 聚合站点中。

- http://perlcast.com/
 Perlcast 是一个 Perl 方面的播客。
- http://blogs.perl.org/
 这是一个 Perl 的博客站点，需要的话请申请自己的账号。
- http://planet.perl.org/
 Planet Perl 是一个 Perl 的 blog 聚合站点。
- http://perlsphere.net/
 Perlsphere 也是一个 Perl 的 blog 聚合站点。

获得帮助

可以从以下几个网站获得 Perl 的帮助。

- http://www.stackoverflow.com/
- http://www.perlmonks.org/
- http://www.nntp.perl.org/group/perl.beginners/是一个针对初学者的邮件列表。
- http://irc.perl.org/
- http://lists.perl.org/列出了许多 Perl 方面的邮件列表，其中大部分都有特定的主题。

www.TopSage.com